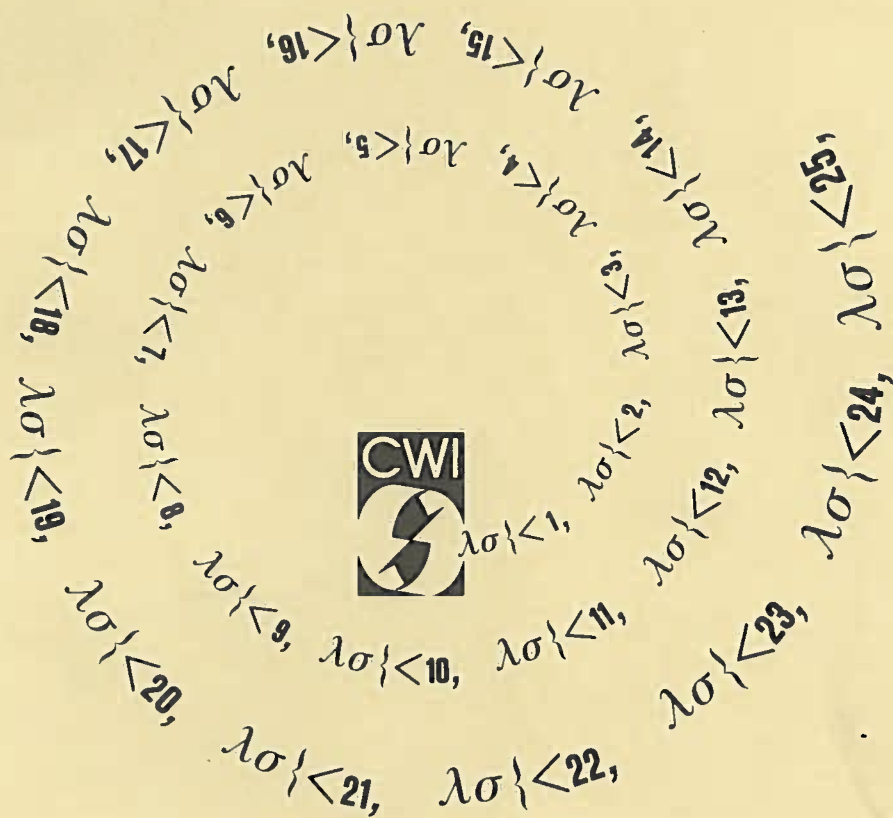
J.W. DE BAKKER, 25 JAAR SEMANTIEK

LIBER AMICORUM



J.W. DE BAKKER, 25 JAAR SEMANTIEK

LIBER AMICORUM

Een uitgave van de Stichting Mathematisch Centrum
ter gelegenheid van het 25-jarig dienstverband van
J.W. de Bakker met het Mathematisch Centrum,
Amsterdam, april 1989

samengesteld door: J.W.Klop
J.-J.Ch.Meijer
J.J.M.M.Rutten
verzameld door: F.Snijders
L.Vasmel
ontwerp omslag: R.T.Baanders
druk: CWI, Amsterdam

Inhoudsopgave

Voorwoord

J.W. de Bakker, D.S. Scott, *A Theory of Programs*, IBM Seminar Vienna , August 1969 — 1

P.H.M. America, *The Practical Importance of Formal Semantics* — 31

K.R. Apt, E.-R. Olderog, *A Note on Disjoint Parallelism* — 41

P.C. Baayen, *Playing at semantics* — 47

R.J.R. Back, *On formal and informal reasoning in program construction* — 53

J.C.M. Baeten, F.W. Vaandrager, *An Algebra for Process Creation* — 57

Jacob de Bakker, *25 jaar voor pap van Jacob* — 83

Jaska de Bakker, *Er was eens een vader in Amsterdam* — 87

Lisa de Bakker, *Voor pap van Lisa* — 89

J.A. Bergstra, J.W. Klop, *BMACP* — 91

E. Best, *Towards Compositional Predicate Transformer Semantics for Concurrent Programs* — 111

D. Bjørner, *Facets of Software Development* — 119

E.K. Blum, *The semantics and complexity of parallel programs for vector computations. Part II* — 133

F.S. de Boer, *A Summary of the Work on the Proof Theory for the Language POOL* — 159

A. de Bruin, E.P. de Vink, *Continuation Semantics for PROLOG with Cut* — 163

M. Dezani-Ciancaglini, R. Hindley, *Intersection Types for Combinatory Logic* — 181

A. Ehrenfeucht, G. Rozenberg, *A Characterization of the State Spaces of Elementary Net Systems* — 193

A. Eliëns, *On the use of semantics: Extending Prolog to a Parallel Object Oriented Language* — 203

P. van Emde Boas, *A compositional semantics for the Turing machine* — 219

N. Francez, *Cooperating-proofs for distributed programs with multi-party interactions* — 229

R.J. van Glabbeek, J.J.M.M. Rutten, *The processes of De Bakker and Zucker represent bisimulation equivalence classes* — 243

R.J. van Glabbeek, W.P. Weijland, *Refinement in branching time semantics* — 247

H.J.M. Goeman, *Towards a theory of (self) applicative communicating processes: a short note* — 253

J. Gruska, *MFCS greetings to Jaco W. de Bakker* — 259

W.H. Hesselink, *The induction rule of De Bakker and Scott* 261

F. Honsell, S. Ronchi Della Rocca, *Qualitative λ-models as Type Assignment Systems* 265

J. Hooman, S. Ramesh, W.P. de Roever, *A Compositional Semantics for Statecharts* 275

A. van der Houwen, P. van der Houwen, *Beste Jaco en Angeline* 289

P. Klint, *Scanner generation for modular regular grammars* 291

J.N. Kok, *Metric Semantics for the Input/Output Behaviour of Sequential Programs* 307

J. van Leeuwen, *Correctness of the two-phase commit protocol* 319

Catrien Lenstra, *Voor Jaco* 329

J.-J.Ch. Meyer, *Correspondenties in semantiek* 331

U. Montanari, *Best wishes* 337

A. Mulder, *Voor Jaco* 339

A. Nijholt, *Views on Parallel Parsing: A preliminary survey* 341

A. Ollongren, *Abstract objects as abstract data types revisited* 359

J. Paredaens, *Uit het Zuiden* 371

A. Pnueli, M. Shalev, *What is in a Step* 373

W. Reisig, *The Decent Philosophers: An exercise in operational semantics of concurrent systems* 401

R.P. van de Riet, *Programmacorrectheid en het college Inleiding Informatica* 407

J.V. Tucker, *Applications of Computability Theory over Abstract Data Types* 421

L. Vasmel, *1964 - MC - J.W. de Bakker - CWI - 1989* 433

J.C. van Vliet, *Over φ's en ψ's* 435

J.I. Zucker, *Congratulations* 437

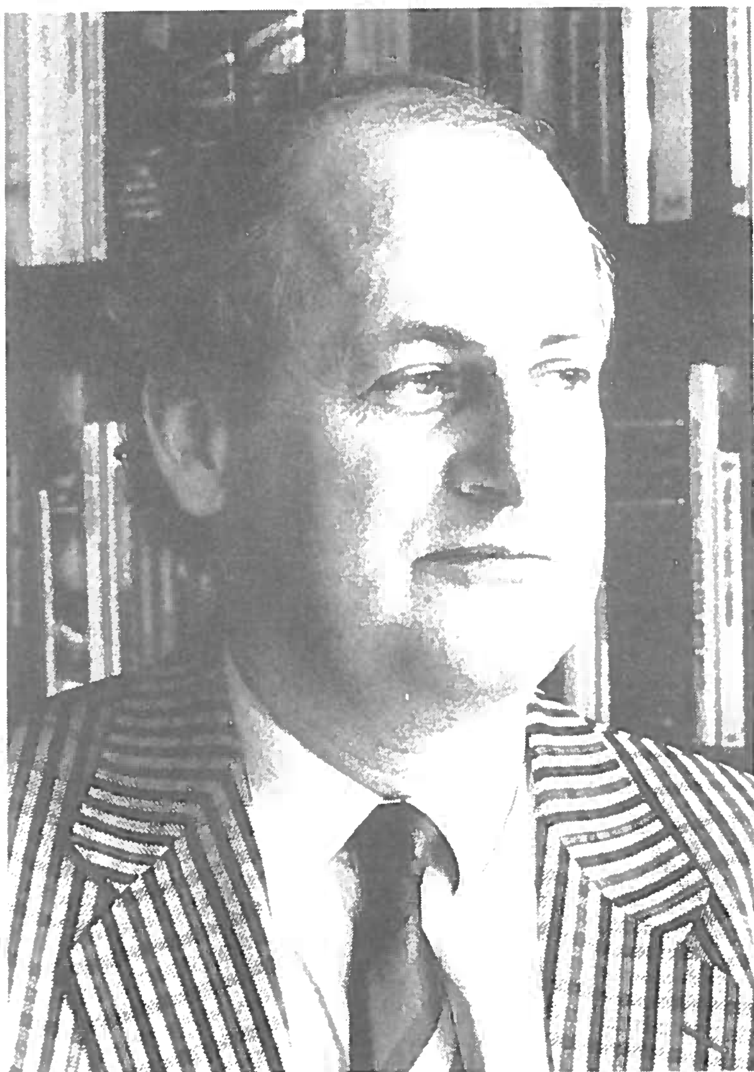
VOORWOORD

Deze bundel is samengesteld ter gelegenheid van het 25-jarige dienstverband van Prof. Dr. J.W. de Bakker met het Centrum voor Wiskunde en Informatica (CWI). In deze 25 jaar heeft Professor de Bakker zich—wat het onderzoek betreft—voornamelijk beziggehouden met ontwikkelingen op het gebied van de semantiek van programmeertalen. De eerste vijf jaar betrof dit in het bijzonder de programmeertalen ALGOL 60 en ALGOL 68, getuige ook de promotie van de jubilaris op het proefschrift "Formal definition of programming languages—with an application to the definition of ALGOL 60", 17 mei 1967, met als promotor Prof. Dr. Ir. A. van Wijngaarden. De laatste twintig jaar heeft hij zich met een grote variëteit van talen en taalconcepten beziggehouden, waarbij de door hem en D.S. Scott ontwikkelde denotationele semantiek een fundamentele rol speelde. We zijn als redactie dan ook bijzonder verheugd dat we deze bundel kunnen openen met een facsimile van het oorspronkelijke en nooit gepubliceerde manuscript van de 'IBM-seminar lecture notes' uit 1969. Dit manuscript gaf een aanzet tot de ontwikkeling van de denotationele semantiek, en heeft een belangrijke rol gespeeld in het verdere onderzoek van De Bakker en de groep van onderzoekers rond hem.

In deze bundel treft u naast het genoemde manuscript een bonte verscheidenheid aan van artikelen en andere zaken, die als volgt tot stand is gekomen. Ter gelegenheid van het 25-jarige jubileum van De Bakker is een feestelijk symposium georganiseerd, op 28 april 1989, waarvoor een aantal vooraanstaande sprekers is uitgenodigd die een belangrijke rol hebben gespeeld in de wetenschappelijke loopbaan van de jubilaris. Aan deze sprekers is tevens verzocht een schriftelijke bijdrage te leveren aan deze bundel. Verder is door middel van een rondschrijven aan collega's, vrienden en bekenden van de jubilaris een uitnodiging uitgegaan een bijdrage te leveren voor dit 'Liber Amicorum'. Hierbij werd de aard van de bijdrage naar eigen keuze gelaten. Zo vindt u naast de technische bijdragen, waarvan het grote aantal ons aangenaam verraste, ook enige meer informele of zelfs ludieke bijdragen.

Wij hopen dat u, lezer, en in het bijzonder de jubilaris, evenveel genoegen zullen scheppen in het lezen van deze collectie als wij hadden in de samenstelling ervan. Tenslotte stellen we er prijs op onze dank uit te spreken aan F. Snijders en Mw. L. Vasmel, voor het vele werk dat zij hebben verricht, in verband met (o.a.) het onderhouden van contacten met al degenen die aan deze bundel hebben bijgedragen en met de organisatie van de feestelijkheden op de symposiumdag; alsmede aan T. Baanders voor de grafische verzorging van de voorplaat (en Joost Kok voor het genereren van ideeën hiervoor), en de reproductieafdeling van het CWI die op zeer korte termijn voor de productie van dit boek zorg droeg.

J.W. Klop
J.-J. Ch. Meyer
J.J.M.M. Rutten



J.W. de Bakker, maart 1989. Foto: J. Rutten

IBM SEMINAR
VIENNA
AUGUST 1969

A THEORY OF PROGRAMS

AN OUTLINE OF JOINT WORK BY
J.W. DE BAKKER AND
DANA SCOTT

MACHINES

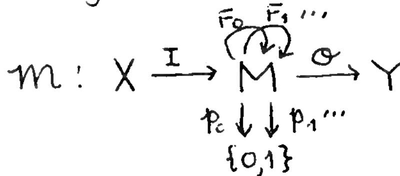
A *machine* is a structure

$$\mathcal{M} = (\mathcal{I}, \mathcal{O}, F_0, p_0, F_1, p_1, \ldots)$$

of *partial* functions for which there exist (uniquely determined) sets X, M, Y such that:

$\mathcal{I} : X \to M$ (the *input* function),
$\mathcal{O} : M \to Y$ (the *output* function),
$F_i : M \to M$ (the *operations*),
$p_j : M \to \{0,1\}$ (the *tests*).

As a diagram we can write:



COMPUTATIONS

A *computation* (from $x \in X$ to $y \in Y$) is a finite sequence

$$\xi_0 \; F_{i_0} \; \xi_1 \; F_{i_1} \; \xi_2 \ldots \xi_{k-1} \; F_{i_{k-1}} \; \xi_k$$

where each $\xi_\ell \in M$ and $\xi_{\ell+1} = F_{i_\ell}(\xi_\ell)$ for $\ell < k$

(and where $\mathcal{I}(x) = \xi_0$ and $\mathcal{O}(\xi_k) = y$.)

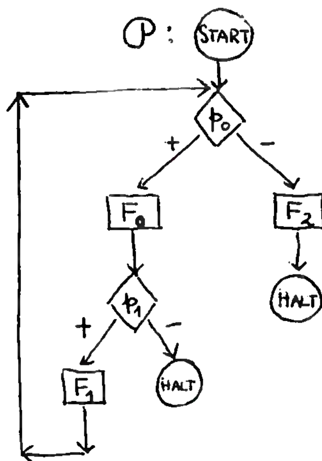
PROGRAMS

Informally a program is a "method" of selecting at most one computation starting from each $x \in X$. Thus a program $\mathcal{P}$ associates to the machine $\mathcal{M}$ a partial function

$$\mathcal{P}(\mathcal{M}) : X \to Y.$$

Programs must be defined "independently" of particular machines, and indeed we can identify the program with this mapping from machines to functions. A program as a mapping must satisfy some general conditions to be mentioned later. First we give some examples.

FLOW DIAGRAMS



It is intuitively clear that, given an arbitrary machine $\mathcal{M}$, this diagram allows us to generate for each $x \in X$ at most one computation, obtained by following the "flow" of the diagram. We say that the diagram (a "syntactical" object) defines a program (a mathematical object.)

PROCEDURES

$$\mathcal{P}: \begin{cases} P_0 \Rightarrow (p_0 \to F_0 ; P_1 \,,\ F_2) \\ P_1 \Rightarrow (p_1 \to F_1 ; P_0 \,,\ E) \end{cases}$$

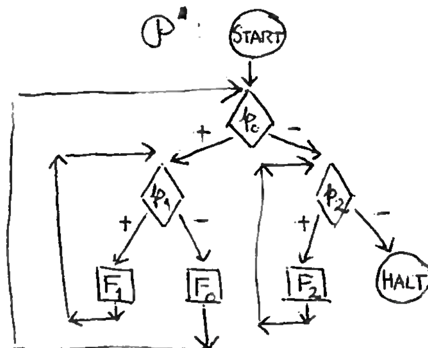
Here we have a system of procedure declarations (where by convention P_0 is the "principal" one.)
$(p \to P, P')$ is the <u>conditional expression</u> ;
$P; P'$ is <u>composition</u> (P followed by P') ; and
E is the symbol for the <u>identity function</u>.
The above system defines the same program as the flow diagram on p. 2.

WHILE STATEMENTS

$$\mathcal{P}': (p_0 * (p_1 * F_1) ; F_0) ; (p_2 * F_2)$$

Read "$(p * P)$" as "<u>while</u> p <u>do</u> P." The same program can be defined by flow diagrams or by procedures:

$\mathcal{P}'$:



$$\mathcal{P}': \begin{cases} P_0 \Rightarrow (p_0 \to P_1 ; P_0 \,,\ P_2) \\ P_1 \Rightarrow (p_1 \to F_1 ; P_1 \,,\ F_0) \\ P_2 \Rightarrow (p_2 \to F_2 ; P_2 \,,\ E) \end{cases}$$

EQUIVALENCES

Two programs $\mathcal{P}$ and $\mathcal{P}'$ are equivalent on a machine $\mathfrak{M}$ iff $\mathcal{P}(\mathfrak{M}) = \mathcal{P}'(\mathfrak{M})$. They are (strongly) equivalent iff they are equivalent on all machines; that is, iff $\mathcal{P} = \mathcal{P}'$. Two flow diagrams (systems of procedures, while statements) are equivalent iff they define equivalent programs.

THEOREM. Every program defined by a while statement is (effectively) equivalent to one defined by a flow diagram, but there is a flow diagram that defines a program not defined by any while statement.

THEOREM. (The same for flow diagrams and procedures.)

THEOREM. It is (effectively) decidable whether two flow diagrams are equivalent.

THEOREM. It is (effectively) decidable whether a system of procedures defines the null program Ω.

PROBLEMS. Is it (effectively) decidable whether two procedures are equivalent? Is it decidable when a procedure is equivalent to some flow diagram? a flow diagram to a while statement?

GENERAL PROPERTIES

We write $\mathcal{M} \subseteq \mathcal{M}'$ iff $\mathcal{M}$ and $\mathcal{M}'$ have the same sets X, M, Y and have operations and tests $F_i \subseteq F_i'$ and $p_j \subseteq p_j'$ for all i, j. (That is, F_i' is consistent with, but more defined than F_i.)

(I) $\mathcal{M} \subseteq \mathcal{M}'$ <u>implies</u> $\mathcal{P}(\mathcal{M}) \subseteq \mathcal{P}(\mathcal{M}')$

(Property (I) can be generalized by defining the notion of <u>morphism</u> $\varphi : \mathcal{M} \to \mathcal{M}'$ between machines. See below)

(II) <u>If</u> $\mathcal{M}_n \subseteq \mathcal{M}_{n+1}$ <u>for</u> $n = 0, 1, 2, \ldots$, <u>then</u>

$$\mathcal{P}\left(\bigcup_{n=0}^{\infty} \mathcal{M}_n\right) = \bigcup_{n=0}^{\infty} \mathcal{P}(\mathcal{M}_n)$$

(Property (II) can be generalized to directed unions (and, no doubt, to direct limits).)

Write $\mathcal{M}^{(n,m)}$ for the result of modifying $\mathcal{M}$ by replacing F_i and p_j by the totally <u>undefined</u> functions for $i \geq n$ and $j \geq m$.

(III) <u>For each program</u> $\mathcal{P}$ <u>there exist</u> n, m <u>such that for all machines</u> $\mathcal{M}$:

$$\mathcal{P}(\mathcal{M}) = \mathcal{P}(\mathcal{M}^{(n,m)}).$$

The above are clear for procedures and (suitably generalized) ought to be taken as "axiomatic" for the notion of program.

CATEGORIES

Let $\mathbb{F}$ be the category whose objects are partial functions $F : X \to Y$ and whose morphisms $\varphi : F \to F'$ are pairs $\varphi = (\varphi^{\downarrow}, \varphi^{\uparrow})$ where

$$\begin{array}{ccc} X & \xrightarrow{F} & Y \\ \varphi^{\downarrow} \downarrow & & \uparrow \varphi^{\uparrow} \\ X' & \xrightarrow[F']{} & Y' \end{array}$$

$\varphi^{\downarrow} : X \to X'$ and
$\varphi^{\uparrow} : Y' \to Y$ and
$F \subseteq \varphi^{\downarrow} ; F' ; \varphi^{\uparrow}$.

(Here ";" denotes composition of relations so that $(F;G)(x) = G(F(x))$ in the case of functions.)

Let $\mathbb{M}$ be the category whose objects are machines and whose morphisms $\varphi : \mathcal{M} \to \mathcal{M}'$ are triples $\varphi = (\varphi^{\downarrow}, \varphi^{\square}, \varphi^{\uparrow})$ where $\varphi^{\downarrow} : X \to X'$ and $\varphi^{\square} : M \to M'$ and $\varphi^{\uparrow} : Y' \to Y$ and where for all i, j :

$$I \subseteq \varphi^{\downarrow} ; I' ; (\varphi^{\square})^{-1}$$

$$O \subseteq \varphi^{\square} ; O' ; \varphi^{\uparrow}$$

$$F_i \subseteq \varphi^{\square} ; F_i' ; (\varphi^{\square})^{-1}$$

$$p_j \subseteq \varphi^{\square} ; p_j'$$

(Here "-1" denotes converse of a relation.)

$$\begin{array}{ccccccc} \mathcal{M} : & X & \xrightarrow{I} & \boxed{M} & \xrightarrow{O} & Y \\ & \varphi^{\downarrow} \downarrow & & \varphi^{\square} \downarrow & & \uparrow \varphi^{\uparrow} \\ \mathcal{M}' : & X' & \xrightarrow{I'} & \boxed{M'} & \xrightarrow{O'} & Y' \end{array}$$

CATEGORIES (CONT.)

Let $\mathcal{P}$ be a program and let $\varphi : \mathcal{M} \to \mathcal{M}'$.
Define $\mathcal{P}(\varphi) = (\varphi^{\downarrow}, \varphi^{\uparrow})$ where $\varphi = (\varphi^{\downarrow}, \varphi^{\square}, \varphi^{\uparrow})$

"THEOREM" If $\varphi : \mathcal{M} \to \mathcal{M}'$ in the category $\mathbb{M}$, then

$$\mathcal{P}(\varphi) : \mathcal{P}(\mathcal{M}) \to \mathcal{P}(\mathcal{M}') \text{ in the category } \mathbb{F}.$$

This is at least clear for programs defined by procedures; it should be taken as "axiomatic" in general. Thus it follows that $\mathcal{P} : \mathbb{M} \to \mathbb{F}$ is a functor between categories. This is the proper generalization of property (I) above. As a functor $\mathcal{P}$ ought to be rather "continuous" in some suitable sense.

If $\mathcal{M} : X \xrightarrow{I} \boxed{M} \xrightarrow{O} Y$, define $[\mathcal{M}] : M \xrightarrow{E} \boxed{M} \xrightarrow{E} M$ to be the machine with the same operations and tests but with I, O, X, Y replaced by E, E, M, M, where E is the identity on M. Then we have the result:

$$(I, E, O) : \mathcal{M} \to [\mathcal{M}] \quad \text{in } \mathbb{M}$$

It was the "obvious correctness" of this relationship that motivated the above definitions. The categories $\mathbb{M}$ and $\mathbb{F}$ require much more study, however, before their usefulness can be determined.

CONTROL DEVICES

By a control device we shall understand a "Boolean" machine:

$$\Gamma : \{0\} \xrightarrow{S} C \xrightarrow{H} \{0,1\}$$

(with operations $G_0, G_1, \ldots$ on C and maps $q_0, q_1, \ldots : C \to \{0,1\}$)

"ABSTRACT" PROGRAMS

Let $N = \{0,1,2,\ldots\}$ and let $\{0,1\}^\infty$ be the set of all partially defined infinite sequences of 0's and 1's. If M is a machine and $\xi \in M$, define

$$p(\xi) = (p_0(\xi), p_1(\xi), \ldots, p_n(\xi), \ldots) \in \{0,1\}^\infty.$$

An "abstract" program $\mathcal{P}^{\Gamma}_{f,g}$ is defined by giving a control device Γ (generally fixed so that a definite class of programs is considered) and two functions

$$f, g : \{0,1\}^\infty \times \{0,1\}^\infty \to N$$

such that the selected computations of $\mathcal{P}^{\Gamma}_{f,g}$ on a machine M (in the notation of p. 1) are those for which there exist (uniquely determined) computations

$$\gamma_0\, G_{j_0}\, \gamma_1\, G_{j_1}\, \gamma_2 \cdots \gamma_{k-1}\, G_{j_{k-1}}\, \gamma_k$$

on the machine Γ where for $l < k$

"ABSTRACT" PROGRAMS (CONT.)

$$i_\ell = f(p(\xi_\ell), q(\gamma_\ell)), \text{ and}$$
$$j_\ell = g(p(\xi_\ell), q(\gamma_\ell)), \text{ and}$$

$$\gamma_0 = S(0), \; H(\gamma_\ell) = 0, \text{ for } \ell < k, \text{ and}$$
$$H(\gamma_k) = 1.$$

Thus S and H control <u>start</u> and <u>halt</u> and f and g tell where to look for the next operation to execute. We need Γ as a "memory" to keep track of where we are in the intermediate stages of "reading" the text of the program definition. Assuming that f and g are "finitely given" (i.e. depend on a fixed bounded number of coordinates of $p(\xi)$ and $q(\xi)$), we can then prove that $\mathcal{P}^{\Gamma}_{f,g}$ is a functor with basic properties (I), (II), (III). (We need some slight <u>consistency</u> conditions on f and g.)

CONJECTURE. There are a few more "nice" properties (like (I), (II), (III)) such that any functor $\mathcal{P}: \mathbb{M} \to \mathbb{F}$ having these properties is of the form $\mathcal{P}^{\Gamma}_{f,g}$.

NOTE: In this abstract setting it is convenient to make the harmless convention that on all machines $\mathfrak{M}$ we have $F_0 = E =$ identity on M

EXAMPLES

The "abstract" version of the <u>flowdiagram</u> uses the control device where

$$C = \{-1, 0, 1, 2, \dots\}$$
$$S(0) = 0$$
$$G_j(\gamma) = j-1$$
$$q_j(\gamma) = \begin{cases} 1 & \text{if } \gamma = j \\ 0 & \text{otherwise} \end{cases}$$
$$H(\gamma) = \begin{cases} 1 & \text{if } \gamma = -1 \\ 0 & \text{otherwise} \end{cases}$$

The "abstract" version of the <u>procedure</u> uses the more general control device where

$$C = \{0, 1, 2, \dots\}^* = \{\sigma_0, \sigma_1, \sigma_2, \dots\}$$
$$S(0) = 0$$
$$G_j(n\gamma) = \sigma_j \gamma$$
$$q_j(n\gamma) = \begin{cases} 1 & \text{if } n = j \\ 0 & \text{otherwise} \end{cases}$$
$$H(\gamma) = \begin{cases} 1 & \text{if } \gamma = \Lambda \\ 0 & \text{otherwise} \end{cases}$$

where Λ is the <u>null sequence</u> and the σ_n represent some (recursive!) enumeration of the finite sequences of integers. (In other words: the control of a procedure computation is in general a <u>push-down store</u>)

DEDUCTIONS

For the time being we restrict attention to <u>procedures</u> and ask how it can be established when two of them are <u>equivalent</u>. In one sense the question is answered because we have given a completely precise definition of the program defined by a procedure with the aid of a certain control device. That answer is not too helpful, because no simple "methods" of proof for proving equivalence are provided by the bare definition. Two different (though related) deductive systems are presented below which might be called the "algebraic" and the "second-order relational" theories. The algebraic method is more efficient for proving <u>equivalences</u>; while the relational method is better for problems of <u>correctness</u>. It is not known whether an algebraic theory can be complete — because if it is, and if it's theorems are recursively enumerable, <u>then</u> we would have a recursive decision method for equivalence. The relational theory <u>is</u> complete — because we use second-order logic — but the theorems are <u>not</u> enumerable.

THE ALGEBRAIC THEORY

LANGUAGE

Lower-case letters are Boolean variables (mostly we use p, q, r) Upper-case are procedure variables, except we use E and Ω as constants. Compound terms are constructed from upper-case letters by these three operations:

$$(p \rightarrow \tau, \sigma) \qquad \tau ; \sigma \qquad \mu X [\tau]$$

where in place of p we can have any Boolean and in place of X any procedure variable. The first is the conditional expression; the second, a composition; and the third has a variable-binding operator μ whose meaning is explained below. Atomic formulas are either equations $\tau = \sigma$ or inclusions $\tau \subseteq \sigma$. Lists $\Phi_0, \Phi_1, \ldots, \Phi_{n-1}$ of atomic formulas are used as a short-hand for the conjunction $[\Phi_0 \wedge \Phi_1 \wedge \cdots \wedge \Phi_{n-1}]$. Theorems are of the form of implications $\Phi \vdash \Psi$ between lists. We use for simplicity in these notes the usual notation $\tau(X, Y)$, $\Phi(X, Z)$, $\Psi(X, \tau(X))$ to indicate (roughly!) free variables and the results of substitutions

A.T. (CONT.)

VALIDITY

Consider an implication $\Phi \vdash \Psi$. Suppose the *free* variables are $p_0, p_1, p_2, \ldots$ and $F_0, F_1, F_2, \ldots$ The implication is *valid* just in case for all sets M and all systems $F_i : M \to M$ and $p_j : M \to \{0,1\}$ of partial functions and predicates on M, *if* Φ is *true* for these, *then* so is Ψ. Now a list is true iff all terms are. An atomic $\tau = \sigma$ is true iff τ and σ denote the *same* function on M into M. An atomic $\tau \subseteq \sigma$ is true iff τ denotes a function *included* in that denoted by σ. Thus, given the values of the free variables we need still only define what is the function *denoted* by a term. E denotes the *identity* function. Ω denotes the *empty* ("undefined") function. *Conditionals* and *compositions* denote functions obtained from the denotations of the parts in the usual way. The special term $\mu X[\tau(X)]$ denotes the *least* function G such that $\tau(G) \subseteq G$. ("Least" in the sense of the partial ordering $\subseteq$.) We shall see below why it always exists, and why it is of interest in connection with procedures

A.T. (CONT.)

THEOREM. The set of valid implications is not recursively enumerable (sorry!)

PROBLEM. Is the set of valid $\vdash \Psi$ recursively enumerable (and hence recursive)?

AXIOMS AND RULES

The axioms and rules for conjunctions and equations are well-known. As for $;\ \subseteq\ E\ \Omega$ we give the axioms of a partially ordered semigroup with unit and zero. We give the usual axioms for conditional expressions and besides:

$$\vdash (p \to X, Y); Z = (p \to X; Z, Y; Z)$$

$$\vdash (p \to X, X) \subseteq X$$

$$X \subseteq X', Y \subseteq Y' \vdash (p \to X, Y) \subseteq (p \to X', Y')$$

$$(p \to X, \Omega) \subseteq Z, (p \to \Omega, Y) \subseteq Z \vdash (p \to X, Y) \subseteq Z$$

(Maybe some others are required?? This point about conditionals is not too clear and needs more study.) For μ we have:

$$Y = \mu X[\tau(X)] \vdash \tau(Y) \subseteq Y \qquad \text{(Axiom)}$$

$$\frac{\Phi \vdash \Psi(\Omega) \qquad \Phi, \Psi(X) \vdash \Psi(\tau(X))}{\Phi \vdash \Psi(\mu X[\tau(X)])} \qquad \text{(Rule)}$$

(In the rule X is not free in Φ.)

A.T. (CONT.)

APPLICATION

Consider the following system of procedures:

$$\mathcal{P}'' \begin{cases} P_0 \Rightarrow \tau_0(P_0, P_1) \\ P_1 \Rightarrow \tau_1(P_1, P_2) \\ P_2 \Rightarrow \tau_2(P_2, P_0) \end{cases}$$

where the τ_i are terms with the P_i and possibly other free variables. The P_i variables, of course, play a special rôle, and the above "rewrite" rules mean that the P_i should be computed as the "least" functions that result from replacing a procedure "call" by the corresponding procedure "body". Hence,

$$P_2 = \mu Z[\tau_2(Z, P_0)] ,$$

and then

$$P_1 = \mu Y[\tau_1(Y, \mu Z[\tau_2(Z, P_0)])] ,$$

and finally

$$P_0 = \mu X[\tau_0(X, \mu Y[\tau_1(Y, \mu Z[\tau_2(Z, X)])])] .$$

Thus the whole program can be defined by the algebraic expression on the right-hand side. Proving equations between expressions, then, *is* proving equivalence of programs.

A.T. (CONT.)

JUSTIFICATION

With a combination of results proved within the system and remarks outside the theory, we will see that the axioms are valid and that the rules preserve validity. Later we give some examples of particular equivalence proofs.

(1) MONOTONICITY

$$X \subseteq X' \vdash \tau(X) \subseteq \tau(X')$$

Proof: The theorem must first be generalized to any number of variables and then proved by induction on the complexity of the term τ. The cases of conditionals and compositions are already assumed as (obviously valid) axioms. For the μ-operator we do a representative special case. Thus assume $\sigma(X, Y)$ monotonic in both variables, and consider $\tau(X)$ to be $\mu Y[\sigma(X, Y)]$. By the first μ-axiom:

$$\vdash \sigma(X', \tau(X')) \subseteq \tau(X'),$$

hence by assumption on σ:

$$X \subseteq X' \vdash \sigma(X, \tau(X')) \subseteq \tau(X')$$

We can again apply the monotonicity of the term σ to derive:

$$X \subseteq X', Y \subseteq \tau(X') \vdash \sigma(X, Y) \subseteq \tau(X').$$

A.T. (CONT.)

Note that $X \subseteq X' \vdash \Omega \subseteq \tau(X')$ is trivial; thus by the <u>rule</u> for the μ-operator we have:

$$X \subseteq X' \vdash \mu Y[\sigma(X,Y)] \subseteq \tau(X') \quad ,$$

which is the desired result for τ.

<u>Discussion</u>. We proved monotonicity by the axioms and rules for μ, but this proof also helps (in part) to establish the validity of these principles. In our calculus the expressions represent <u>monotonic operations</u> on partial functions. It is well-known that such operations have <u>minimal fixed points</u>. Speaking informally:

$$\mu X[\tau(X)] = \bigcap\{X : \tau(X) \subseteq X\}$$

$$= \bigcup_{n=0}^{\infty} \tau^n(\Omega)$$

where $\tau^n(\Omega) = \overbrace{\tau(\tau(\cdots \tau}^{n\text{-times}}(\Omega)\cdots))$. The second equation, which justifies the special case of the rule used in the proof of (1), is correct because the operations are also <u>continuous</u> in this sense (speaking outside the theory):

$$\tau\left(\bigcup_{n=0}^{\infty} X_n\right) = \bigcup_{n=0}^{\infty} \tau(X_n)$$

whenever $X_0 \subseteq X_1 \subseteq \cdots \subseteq X_n \subseteq \cdots$. This is clear for conditionals and compositions, but again must be established for μ-s. Once

A.T. (CONT.)

continuity is understood, the validity of the full rule for μ (which we may call the <u>induction principle</u>) follows easily.

(2) FIXED POINT PROPERTIES

$$Y = \mu X[\tau(X)] \vdash \tau(Y) = Y \quad \text{and}$$

$$\tau(Y) \subseteq Y \vdash \mu X[\tau(X)] \subseteq Y$$

The proof uses monotonicity and induction as in the proof of (1).

(3) SYSTEMS OF FIXED POINTS

$$X = \mu X[\tau(X, \mu Y[\sigma(X,Y)])],$$

$$Y = \mu Y[\sigma(X,Y)],$$

$$\tau(X', Y') \subseteq X', \ \sigma(X', Y') \subseteq Y' \vdash$$

$$\tau(X,Y) \subseteq X \subseteq X', \ \sigma(X,Y) \subseteq Y \subseteq Y'$$

<u>Proof</u>: "assume" the four hypotheses. Then $\tau(X, \mu Y[\sigma(X,Y)]) \subseteq X$ and so $\tau(X,Y) \subseteq X$. Also $\sigma(X,Y) \subseteq Y$. Let $Y'' = \mu Y[\sigma(X',Y)]$. Then $Y'' \subseteq Y'$ and so $\tau(X', Y'') \subseteq X'$. But then $X \subseteq X'$ and so $\sigma(X, Y') \subseteq Y'$. Finally $Y \subseteq Y'$.

<u>Discussion</u>. The above theorems on minimal fixed points can be extended to systems with <u>any number</u> of procedure declarations.

A.T. (CONT.)

EXAMPLES

We define *while* as $(p * F) = \mu X[(p \to F; X, E)]$

(i) $(p * F); G = \mu Y[(p \to F; Y, G)]$

Proof. By definition $(p * F) = (p \to F; (p * F), E)$

Hence $(p * F); G = (p \to F; (p * F); G, G)$

Therefore $\mu Y[(p \to F; Y, G)] \subseteq (p * F); G$.

To prove the opposite inclusion, first let $Y = \mu Y[(p \to F; Y, G)]$. By induction it is enough to show $\Omega; G \subseteq Y$ (obvious!) and $X; G \subseteq Y \vdash (p \to F; X, E); G \subseteq Y$. So assume $X; G \subseteq Y$. But $(p \to F; Y, G) \subseteq Y$.

Thus $(p \to F; X, E); G = (p \to F; X; G, G) \subseteq Y$

(ii) $p * (p * F) = p * (F; p * F)$

Proof. Let L be the left-hand side and R the right.

Then $L = (p \to (p * F); L, E)$

$= (p \to (p \to F; (p * F), E); L, E)$

$= (p \to F; (p * F); L, E)$

Thus $R \subseteq L$

Next $R = (p \to F; (p * F); R, E)$

$= (p \to (p \to F; (p * F), E); R, E)$

$= (p \to (p * F); R, E)$

Thus $L \subseteq R$ and $L = R$ follows.

A.T. (CONT)

EXAMPLES (CONT)

(iii) $(p * F); (p \to G, H) = (p * F); H$

(iv) $(p * F); (p * G) = p * F$

(v) $p * (p * F) = p * F$

all of these follow easily from (i) and the method shown for (ii).

THE RELATIONAL THEORY

Functions are, after all, relations. Thus it must be possible to axiomatize the functions defined by program expressions, hopefully without the minute detail of the definitions involving control devices which are closer to the ideas of implimentation. We do this for procedures

LANGUAGE

We use simply a standard second-order predicate calculus with equality and notational conventions corresponding to our language of procedure declarations. In particular we employ the following styles of variables and non-logical constants:

R.T. (CONT.)

Individual variables $\xi, \eta, \zeta, \xi', \eta', \ldots$

1-place predicate constants.

$$p, \bar{p}, q, \bar{q}, r, \bar{r}, \ldots$$

Binary relation variables: $R, S, T, X, Y, Z, \ldots$

Binary relation constants:

$$E, \Omega, P_0, P_1, P_2, \ldots$$

Relational operations: $(R;S)$ $(p \to R, S)$

Atomic formulas: equations plus

$$R \subseteq S \qquad p(\xi) \qquad \xi R \eta$$

(where in place of R and S we can have relational terms and in place of p the other $\bar{p}, q, \bar{q}, \ldots$ and in place of ξ, η any individual variables.)

VALIDITY AND DEDUCTION

This is the usual notion from second-order logic; we do, however, have a few non-logical axioms to suit the application to procedures. Remember that the valid formulas are not recursively enumerable in second-order logic

R.T. (CONT.)

GENERAL AXIOMS

We assume once and for all the following definitions and axioms:

(1) $\neg \exists \xi [p(\xi) \wedge \bar{p}(\xi)]$ (sim. for $q, r, \ldots$)

(2) $\forall \xi, \eta [\xi E \eta \leftrightarrow \xi = \eta]$

(3) $\neg \exists \xi, \eta [\xi \Omega \eta]$

(4) $\forall \xi, \eta [\xi (R;S) \eta \leftrightarrow \exists \zeta [\xi R \zeta \wedge \zeta S \eta]]$

(5) $\forall \xi, \eta [\xi (p \to R, S) \eta \leftrightarrow$
$[[p(\xi) \wedge \xi R \eta] \vee [\bar{p}(\xi) \wedge \xi S \eta]]]$

(6) $R \subseteq S \leftrightarrow \forall \xi, \eta [\xi R \eta \to \xi S \eta]$

(where the R, S should be universally quantified

The meaning of the axioms is clear except maybe for (1). Here the <u>pair</u> $p, \bar{p}$ is to represent a partial predicate (by convention all predicates in logic are total.) The p is the <u>true</u> part and the $\bar{p}$ the <u>false</u> part. They must be disjoint – but that is the only requirement. We do not need any similar tricks for partial functions since they are just relations in a straight-forward way.

R. T. (CONT.)

SPECIAL AXIOMS

Consider a program $\mathcal{P}$ defined by a system of declarations:

$$\mathcal{P}\begin{cases} P_0 \Rightarrow \tau_0(P_0,\dots,P_n) \\ \vdots \\ P_n \Rightarrow \tau_n(P_0,\dots,P_n) \end{cases}$$

Corresponding to this system we have:

$$(i_{\mathcal{P}})\quad [\tau_0(P_0,\dots,P_n) \subseteq P_0 \wedge \tau_1(P_0,\dots,P_n) \subseteq P_1 \wedge \dots \wedge \tau_n(P_0,\dots,P_n) \subseteq P_n]$$

$$(ii_{\mathcal{P}})\quad \forall R\,[\bigwedge_{i=0}^{n}[\tau_i(R_0,\dots,R_n) \subseteq R_i] \rightarrow \bigwedge_{i=0}^{n}[P_i \subseteq R_i]]$$

JUSTIFICATION

Given a system of procedures, we have merely assumed – in second-order language – that the P_i are the least relations where $\tau_i(P_0,\dots,P_n) \subseteq P_i$, $i=0,1,\dots,n$. This was the same idea as for the algebraic theory – except here we do not have the μ-operator. We could introduce it, and then all the algebraic principles could be proved from the second-order axioms.

R. T. (CONT.)

REMARK.

Assuming that the free variables of the procedures are $F_0, F_1, F_2, \ldots$ (our usual convention) we might want to assume that they are partial functions (our usual convention). We should have, then, as general axioms:

$$(1') \quad \forall \xi, \eta, \eta' [\xi F_i \eta \wedge \xi F_i \eta' \rightarrow \eta = \eta']$$

These axioms do not seem to make too much difference, however.

APPLICATIONS

Suppose $\mathcal{P}$ and $\mathcal{P}'$ are two programs defined by procedures (where, say, in the second system we use constants $P_0', P_1', \ldots$) The equivalence problem is to deduce, therefore, the statement $P_0 = P_0'$ from all the axioms combined. The use of second-order deductions does not seem, however, any more convenient than the algebraic method for such equivalence. The point of the second-order system lies, rather, in the fact that more and different kinds of problems can be expressed within it.

R.T. (CONT.)

WHILE STATEMENTS

For the sake of illustration we restrict attention to *while statements*. These are special procedures, and we may as well introduce them by an additional *operation* on relations into the language: $(p * R)$. Our axioms (i_3) and (ii_3) then become:

$$(i_*) \quad (p \to F; (p * F), E) \subseteq (p * F)$$

$$(ii_*) \quad \forall R [(p \to F; R, E) \subseteq R \to (p * F) \subseteq R]$$

(where F is to be taken as a variable.)

EXAMPLE

Axiom (ii_*) seems to be a *special* case of our algebraic induction axiom — *but* it is much stronger in view of the quantifier $\forall R$. The reason is that we may specialize R to *any* relation, not just those that can be defined by terms using the operations we have given a notation for. As an illustration we prove

$$(p \to F; R, G) \subseteq R \to (p * F); G \subseteq R$$

(compare example (1) on p. 19.)

R.T. (cont.)

EXAMPLE (cont.)

<u>Proof</u>: With individual variables the conclusion requires:

$$\forall \xi, \eta, \zeta \, [\, \xi (p * F) \eta \wedge \eta G \zeta \to \xi R \zeta \,] \, ;$$

equivalently:

$$\forall \xi, \eta \, [\, \xi (p * F) \eta \to \forall \zeta \, [\eta G \zeta \to \xi R \zeta \,]\,] .$$

This suggests we introduce the relation S such that (and here we use second-order logic to know the S exists):

$$\forall \xi, \eta \, [\, \xi S \eta \leftrightarrow \forall \zeta \, [\eta G \zeta \to \xi R \zeta \,]\,] .$$

<u>Now</u> the problem is to show:

$$(p * F) \subseteq S .$$

In view of axiom (ii_*) it is sufficient to prove:

$$(p \to F ; S , E) \subseteq S .$$

This requires two cases:

(a) $\quad p(\xi) \wedge \xi (F ; S) \eta \to \xi S \eta$

(b) $\quad \bar{p}(\xi) \to \xi S \xi$.

For (a) assume $p(\xi)$ and $\xi F \zeta$ and $\zeta S \eta$. We want to show $\xi S \eta$. So assume in addition $\eta G \zeta'$ and show $\xi R \zeta'$. But we know by hypothesis $(p \to F ; R , G) \subseteq R$.

R.T. (cont)

EXAMPLE (cont.)

Also, since $\eta\, G\, \zeta'$ holds and $\zeta S \eta$, we have $\zeta R \zeta'$ Hence $\xi (F;R) \zeta'$. But then $\xi R \zeta'$ follows at once. Case (b) is even easier.

REMARK

From the work of Hoare we can find a simplification of axiom (ii_*); namely, it can be replaced by the combination of these two axioms:

$$(ii_*') \quad \forall \xi,\eta\, [\xi (p*F) \eta \rightarrow \bar{p}(\eta)]$$

$$(ii_*'') \quad \forall u\, [\forall \xi,\eta\, [u(\xi) \wedge p(\xi) \wedge \xi F \eta \rightarrow u(\eta)] \rightarrow \forall \xi,\eta\, [u(\xi) \wedge \xi (p*F) \eta \rightarrow u(\eta)]]$$

In particular (ii_*'') reads more like the arithmetic induction axiom. (Here u is a unary predicate <u>variable</u> and thus second-order form with $\forall u$ is equivalent to the earlier form with $\forall R$.) A similar simplification of the axiom (ii_p) for procedures is not yet apparent.

R.T. (cont.)

CORRECTNESS

In order to prove programs "correct", Floyd, and later Hoare, have been working with the idea of taking (a part of) a program (relation) P and thinking of desirable properties u and v such that if you <u>enter</u> P in u, you exit P in v. Hoare writes (more or less).

$$u \{P\} v$$

In our second-order logical notation we would write more fully:

$$\forall \xi, \eta \, [\, u(\xi) \wedge \xi P \eta \rightarrow v(\eta) \,]$$

This point of view has several advantages:

1) It is part of a well-known logical system.
2) $u\{P\}v$ becomes an ordinary proposition that can be negated, etc., etc.
3) Floyd's "logical" rules become obvious
4) Floyd's <u>incorrect</u> existential rule is avoided.

CONCLUSIONS

Starting from an intuitively correct idea of a machine, we explained and developed a theory of programming concepts (mainly <u>procedures</u>) This work <u>could</u> be extended by investigating more powerful control devices. But that is probably <u>not</u> a good idea at this level of abstraction What is needed is a <u>more refined</u> model of a machine. In the work above we have treated each "state vector" $\xi \in M$ as a whole – and it is remarkable how many sensible things there are say even so. But in "real life" a vector ξ has <u>components</u>, and these are generally modified more or less independently during computation. This idea should be introduced into the model; <u>then</u> in the programs we will want to use <u>assignment statements</u> to modify the coordinates. This means that the operations on M, the various F_i and p_j, are being <u>analized</u> instead of being treated as "wholes". Along with

Conclusions (cont.)

these problems, we will also want to treat <u>scope</u> of "variables". One way to do this is to make the components of a state vector into <u>push-down stores</u>. But there are so many problems of "reference" that we might want to use Strachey's method of L- and R-values and LUP's. If we can do this, we will then want to <u>isolate</u> general properties of the programs (<u>defined</u> already with the aid of the machine model) in order to <u>organize</u> our <u>deductions</u> more clearly (the so-called "axiomatic" method.) At one level of abstraction this has all been illustrated above. To really carry out the proposal for the "real life" situation is a big, big "program". The outlines seem clear, however, <u>and</u> we should be able to do it in such a way that it actually <u>refines</u> the present theory and keeps all the "abstract" results intact.

The Practical Importance of Formal Semantics

Pierre America
Philips Research Laboratories
Eindhoven, the Netherlands

Abstract

In this paper some possible directions are sketched along which formal semantics can be applied to practical problems. The most promising application areas are language implementation, language design, and program design. It is argued that in order to exploit these possibilities, a closer cooperation between semanticists and practitioners in the above areas is needed.

1 Introduction

The study of formal semantics of programming languages is often considered to be a field of little practical relevance. This point of view is taken by many people that are not familiar with these techniques. Sometimes they are scared off by the first mathematical formula in the text. Others read patiently through all the definitions and even come to understand them, but finally reach the conclusion that the semantics of a real program is a far too complex mathematical object for anyone to understand, and that therefore the whole field cannot lead to any useful result.

Even sadder is the viewpoint of many researchers in the field itself. Many of them have little or no interest for the practical relevance of what they are doing and are completely satisfied with published papers as the sole outcome of their research. Others have a wrong impression of what 'practical relevance' means because they have little or no contact with the people that actually build computer systems and for whom their results could be valuable.

In this paper, I shall try to demonstrate that formal semantic techniques can be useful in many fields of computer science. We shall see that it is not easy to apply these techniques, and that the only possibility is for semanticists and 'practical' people to work closely together, so that each gets a better understanding of the other's problems and capabilities.

2 Semantic formalisms

There are several broad categories in which formal semantic techniques can be divided: operational, denotational, and axiomatic semantics (for a more extensive overview, see

[Pag81]). Sometimes it is said that operational semantics is useful for the language implementor, denotational semantics for the language designer, and axiomatic semantics for the one who writes programs in the language. We shall see below that this is not the complete truth.

Broadly speaking, one can say that *operational semantics* involves some kind of abstract machine and it describes the meaning of the program in terms of the actions that this machine performs. The nature of this abstract machine differs considerably between the several operational formalisms. In VDL [Weg72], the machine and the instructions it executes both have the structure of a tree, where labels can be attached to nodes and edges. The operational semantics presented in [Bak80] uses states, which are functions from variables to values, and state transformations, functions from states to states, which describe the execution of a complete statement. The sequences of states resulting in this way come very close to what in other contexts is called denotational semantics.

A very popular approach is the use of *transition systems* in the 'Structured Operational Semantics' (SOS) style of [HP79,Plo81,Plo83]. Here the meaning of a program is expressed as a sequence of *transitions* between *configurations*. A configuration typically consists of a state, mapping variables to values, plus that part of the program that is still to be executed. A *transition relation* describes the collection of all possible transitions from one configuration to another. Typically this relation is defined inductively by using axioms and rules. Because of the fact that the program to be executed is contained in the configurations, the axioms and rules defining the transition relation can be closely connected to the syntactic structure of the programming language.

The essence of *denotational semantics* lies in the principle of *compositionality:* The semantics takes the form of a function that assigns a meaning, an element of some mathematical domain, to each individual language construct. The principle of compositionality says that this function should be defined in such a way that the meaning of a composite construct does not depend on the form of the constituent constructs, but only on their meaning.

There are several kinds of mathematical domains being used as the ranges of these meaning functions. In most instances some kind of limit construction is necessary to describe infinite behaviour and the structure of the domain should enable such a limit construction. Most forms of denotational semantics use some order-theoretic structure and among these complete partial orders (CPOs, see. e.g., [Bak80]) and several kinds of lattices [Sco76] are the most popular. A quite different approach is the use of complete metric spaces [BZ82].

Finally, instead of directly assigning a meaning to a program, *axiomatic semantics* gives a description of the constructs in a programming language by providing axioms that are satisfied by these constructs. The most popular formalism to express these axioms is Hoare logic [Hoa69]. Here a program or statement is described by two logical assertions: a precondition, describing the state of the system before executing the program, and a postcondition, describing the state after execution. Using such an axiomatic description of the programming language, it is possible, at least in principle, to prove the correctness of a program with respect to a specification.

3 Language implementation

Intuitively the most obvious relationship between formal semantics and practical applications is concerned with the implementation of programming languages. One would hope that having taken all the trouble to give a formal semantics to a programming language, it would be possible to arrive quickly at an implementation for this language. However, things are not so easy as that.

The most obvious candidate for transformation into implementations is operational semantics: this type of semantics already takes the form of describing the actions performed by an abstract machine. Unfortunately, it is not so easy to relate the abstract machine model used by most versions of operational semantics to a concrete machine. Concrete machine architectures are always designed with the objective to implement them cheaply and efficiently in hardware. Moreover, their finiteness imposes many limits on their power, which of course do not apply to abstract machines.

At the moment, I am not aware of any attempt to implement the tree-like structures of the VDL abstract machines on ordinary hardware. For SOS-style transition systems [HP79,Plo81,Plo83], however, there is an interesting possibility: It is relatively easy to write the axioms and rules that describe the transition relation in the form of Horn clauses. In this way they could in principle be interpreted by a system such as Prolog. There are, of course, a few problems with this approach. The most obvious one pertains to the performance of this implementation, which of course can never be very good, but it may be sufficient for a first prototype implementation of a new language. The second one is more fundamental: the Prolog system itself is not complete, in the sense of being able to find all the solutions of the given goal to which the axioms give rise. Therefore the order and the form in which the axioms are written are very important for the execution of the program, and this effectively prohibits a fully automatic implementation along these lines. There exist complete theorem provers that are able to deal with Horn clauses, but these are orders of magnitude slower than the best Prolog implementations and they use much more memory.

The last problem is common to all semantics-based implementation techniques: how to deal with nondeterminism? A transition system may allow several different transitions from a given configuration. In principle, a single program may give rise to many possible transition sequences among which one should be chosen. Sometimes the semantics imposes requirements on the whole transition sequence, for example that it should be fair, or that it should not lead to a dead end. It is clear that this kind of extra requirements is extremely difficult to deal with in the above approach. Prolog traverses its search space in a depth-first way, which is not fair in most cases. Backtracking out of 'wrong' transition sequences is only possible if they are finite. The best solution to this problem would be if the original transition system were formulated in such a way that all its resulting transition sequences are acceptable.

Despite all appearances, denotational semantics may provide at least as good a basis for the automatic generation of language implementations as operational semantics. This may not be the case for the direct form of denotational semantics as described in [Bak80, chapters 1–9] and [Sto77, chapters 1–10], because the functions from states to

states resulting from these definitions are very cumbersome to compute. However, for semantic definitions using *continuations* opportunities appear to be better. The technique of continuations can best be summarized by saying that in defining the meaning of a statement or expression one considers the meaning of the following statements and expressions as a parameter. The output of the semantic function is then the meaning of the statement/expression under consideration plus all the following ones. This combined meaning need not be a function from states to states; it can equally well be a sequence of items to be written to the output file, for instance. For a nice introduction to continuation semantics, see [Gor79].

Once an abstract structure for the syntactic constructs in a language has been fixed (this could form the output of a parser), it is very easy to write the semantic definitions in a functional programming language (e.g., Miranda [Tur85]). If moreover the semantics makes clever use of continuations and the functional language is evaluated in a 'lazy' way, then the expression giving the semantics of a program can be executed directly, giving the output of the program as soon as it has been computed. This even works for nonterminating programs. I have done this experiment a few years ago with the continuation semantics for the language SMALL in [Gor79], using an interpreted SASL implementation [Tur79]. The resulting performance was acceptable for a prototype implementation and the recent advances in the implementation of functional languages [PJ87] would probably lead to an improvement by several orders of magnitude. Other approaches roughly along this line have been done before [Mos76].

Nevertheless, the performance that can be achieved by directly executing traditional denotational descriptions in this way is still much less than the performance exhibited by ordinary hand-written compilers. The most important reason for this is that in the abovementioned approach, operations that can be efficiently performed by computer hardware, such as reading or changing the contents of storage cells, are first mapped to relatively complicated mathematical notions (such as a state: a function from variable names to values), which must then be mapped back again to a concrete computer architecture. This detour can be avoided by splitting the semantic description in several levels. The lowest level provides those basic operations provided by the underlying architecture. It should comprise both a precise mathematical description and a direct translation to machine code. The higher levels can now make use of these basic operations. In this way the higher level description can be shorter, easier to understand, and easier to translate into an efficient implementation. This approach is being explored at several places [MW86b,LP87], with a resulting performance comparable with commercial, hand-written implementations.

Automatic generation of implementations is not the only way in which formal semantics can help in language implementation. Also for hand-crafted implementations it is often very helpful to relate them to a formally defined semantics. Here the basic problem is ensuring the *correctness* of the implementation. A prerequisite for the formal approach is, of course, the existence of some formal semantic description of the programming language (see section 4).

To my knowledge, no complete implementation of a realistic programming language has yet been verified completely formally. But less ambitious goal are certainly

reachable. Among others, a formal semantic model can be used to justify a certain implementation or optimization technique. For example, in [Vaa86] it has been shown that in the implementation of a parallel object-oriented language, the use of message queues leads to a correct implementation with respect to the semantics, which imposes certain fairness requirements on the execution of a program. Moreover, it is shown that certain built-in data types, such as integers, that are conceptually modelled in the language as full-fledged objects, can in fact be implemented in the traditional way without any harm to the semantics. More sophisticated optimizations form the subject of current research. For another example, we refer to the techniques used for strictness analysis in functional programming languages, which are justified by a formal semantic study using abstract interpretation [HBPJ88].

4 Language design

Another activity where formal semantic methods can be applied fruitfully is the design of programming languages. The main purpose here is to provide a formalism in which the outcome of this language design process can be represented. Unfortunately, most descriptions of programming languages are informal, using natural language (e.g., [ANS83,BSI82]. It turns out that, whatever care is taken to make such a description precise and unambiguous, there always remain some points that are open for several different interpretations [Spe82,WSH77]. If we want to describe a certain aspect of a programming language precisely, without any possibility for misinterpretations, the only option is giving a *formal* description.

For describing the syntax of the language, the Backus-Naur form (BNF) has been in common use for many years, sometimes supplemented by much less popular formalisms, such as attribute grammars or two-level grammars, to capture the non-context-free aspects of the syntax. All these formalisms do is defining what is a legal program in the language and what is not; they do not tell us what a program means.

Complete formal semantic descriptions of commonly used programming languages are very rare (but see, e.g., [AH82,MW86a]). It is certainly not easy to give a formal description, be it operational, denotational, axiomatic, or otherwise, of programming languages like Pascal, C, or Ada. Sometimes it is said that the reason for this is that the formalisms for specifying the semantics of a language are not sufficiently developed. This is only partly true. Another reason, at least equally important, is that many of the languages that are currently in use have been designed by language implementors, with a clear view of the implementation of the language in mind, but with far less attention for an independent description of the exact meaning of programs. Some language designers even think that they have sufficiently defined the language when they have constructed one implementation for it.

The lack of appreciation of formal semantic techniques in circles of language designers is even more unfortunate because these techniques can do much more than just provide an unambiguous notation for expressing the meaning of programs. Trying to give a formal semantics for a language is a very good way of detecting weak points in the language design itself. 'Insecurities' as mentioned in papers like [Spe82,WSH77]

are pinpointed very quickly as soon as one tries to describe the meaning of programs in a formal way. At this point it would be possible to give a long range of examples of language design errors that could have been avoided. Of course, the presence of such problems can be demonstrated effectively by giving suitable example programs, but their absence can only be proved by a formal semantics.

The worst form of language design errors are the completely unintentional ones, where the language behaves in a way that is not expected and even less desired by its designer. For example, the Eiffel programming language was claimed to have a completely safe static type checking mechanism [Mey87], but closer analysis shows that this is not true [Coo89]. this kind of errors can be detected by defining a formal semantics for the language and rigorously proving the desired property. Sometimes it suffices to construct a simplified model, which concentrates on the particular aspect under concern (in the case of Eiffel's type system, the analysis in [Car88] clearly indicates some of the problems and possible solutions).

Another kind of errors is the result of conscious decisions by the language designer, where these decisions are nevertheless undesirable because they harm the readability, writability, or implementability of the language. It can be said that operational semantics, denotational semantics, and axiomatic semantics, in this order, are increasingly sensitive instruments for detecting this kind of problems in language design. A good rule of thumb says that there is such a problem whenever the description of a language feature that is considered to be unessential requires a special adaptation of the overall semantic model used to describe the language. For example, in a language like Pascal, the addition of a goto statement requires a substantial adaptation of the denotational description (see, e.g., [Bak80, chapter 10]). Another common phenomenon is the presence, in a language that is meant to be deterministic, of 'unspecified' behaviour in certain circumstances. A denotational description of such a language would have to be adapted to some form of nondeterminism, which indicates that there is a problem. In practice, such unspecified behaviour can form an important obstacle in porting programs from one implementation to another.

While for many programming languages the goal of a clean denotational semantics has not been reached (in many cases it has not even be considered!), the actual goal should be a satisfactory *axiomatic* semantics, which is even more difficult to attain. As we shall in the next section, the usability of a programming language for the systematic construction of reliable programs will depend increasingly on the possibility of applying formal techniques during program design, and for this a good formal proof system for the language is a necessary prerequisite.

5 Program design

Finally, an important area where formal semantic techniques might be applied is in the construction of programs. It is often said that formal semantics, in particular of the operational or denotational kind, is useless because the meaning of a realistic program is such a complex mathematical object that nobody can understand it. The latter statement is certainly true in most cases, but nevertheless the conclusion is not

justified.

A formal definition of a programming language, including both syntax and semantics, could be a concise and unambiguous reference document that can answer all questions about the language, in the same way as a BNF description is a generally accepted and appreciated form of describing the context-free syntax of a language. The question is whether this document is readable by a programmer without several years of studying complicated mathematics. For many forms of denotational semantics this is indeed a problem, but not so severe as one would think: Often the 'complicated mathematics' is only needed to justify the definitions, e.g., by demonstrating that a certain equation indeed has a unique (or otherwise distinguished) solution. In reading the semantics it is mostly possible to ignore these mathematical issues completely (as is demonstrated, for example, in [Gor79]). However, it must be admitted that even then denotational descriptions of programming languages are relatively hard to understand.

This is not necessarily the case for other forms of formal semantics, however. For example, the SOS style of operational semantics [HP79,Plo81,Plo83] employs a set of axioms and rules for defining the set of possible transitions between configurations. Broadly speaking, an axiom describes the basic functionality of a single kind of statement or expression, whereas a rule describes in which way the constituents of a composite construct are evaluated. Because of the strong connection with the syntactic structure and the fact that the mathematics involved is usually not very complicated, this form of semantics leads to descriptions that are very easy to read, even for non-specialists. For example, the operational semantics for the language POOL [ABKR86] turned out to be relatively easy to understand to prospective programmer and implementors of the language.

The form of semantics that is directed most specifically towards the programmer, axiomatic semantics, has not yet become very successful, unfortunately. The reason for this is probably that it is even more difficult to give a clear axiomatic semantics to a language that was not designed with this in mind, than to provide it with a denotational semantics. As most commonly used languages do not even lend themselves to a reasonable denotational semantics, a complete axiomatic semantics is completely out of the question. For some realistic programming languages an attempt has been made to describe them axiomatically, but either the correctness and completeness of the description was an open question [LGH*78] or only a part of the language was described [HW73]. (For the programming language POOL an axiomatic description has been given [Boe89], but it must be admitted that this is not easy to read for an average programmer.)

Nevertheless, formal techniques are playing an increasingly important role in the design of software systems, especially in situations where the reliability of the software is vital. Automatic support for many of the manipulations of the formal objects involved is gradually becoming available. It is clear that at the basis of these formal techniques there should be a formal axiomatic description of the programming language, or in other words, a formal proof system. It is not absolutely necessary that this proof system is *complete*, in the sense that it can prove all correct assertions about any program, but it should, of course, deal with all aspects of the programming

language, and it absolutely must be *sound*, which means that everything that can be proved in the proof system is actually true.

In the mean time, even if a complete axiomatic semantics of the programming language in use is not available, a certain knowledge of the techniques of formal program verification can give the programmer excellent guidelines in the design of his program. Reasoning quasi-formally in terms of assertions describing the state at certain points of the program can provide the support necessary to get subtle pieces of code correct.

6 Conclusions

We have seen that formal semantic techniques can be usefully applied in language implementation, language design, and program design. It turned out that the traditional opinion of operational semantics being mostly useful for language implementors, denotational semantics for language designers, and axiomatic semantics for programmers, does no longer correctly represent the state of affairs: Denotational semantics provides a good starting point for automatic language implementations, SOS-style operational semantics can provide the programmer with a concise and accurate description of what the language constructs do, and a clean axiomatic description should be one of the foremost goals in language design.

There are many opportunities for applying formal semantics, but it is clear that in order to use them, much more cooperation is needed between researchers in the field of semantics and the people active in the areas of language implementation, language design, and programming. As long as semantics is being considered as a purely academic discipline with no relevance to the real world, it will be just that, and it takes more than just proclaiming the practical importance of semantics in order to change this situation.

References

[ABKR86] Pierre America, Jaco de Bakker, Joost N. Kok, and Jan Rutten. Operational semantics of a parallel object-oriented language. In *Conference Record of the 13th Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 13–15, 1986, 194–208.

[AH82] Derek Andrews and Wolfgang Henhapl. Pascal. In Dines Bjørner and Cliff B. Jones, editors, *Formal Specification and Software Development*, 175–251, Prentice-Hall International, Englewood Cliffs, New Jersey, 1982.

[ANS83] ANSI. *The Programming Language Ada Reference Manual*, ANSI/MIL-STD-1815A-1983, approved 17 February 1983. Lecture Notes in Computer Science 155, Springer-Verlag, 1983.

[Bak80] J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.

[Boe89] Frank S. de Boer. *A proof system for POOL.* Technical Report, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, 1989. To appear.

[BSI82] BSI. *Specification for the computer programming language Pascal.* Standard BS 6192, British Standards Institution, Herts, United Kingdom, 1982.

[BZ82] J. W. de Bakker and J. I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Coo89] William Cook. A proposal for making Eiffel type-safe. In *ECOOP'89: European Conference on Object-Oriented Programming*, Nottingham, England, July 10–14, 1989. To appear.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[HBPJ88] Chris Hankin, Geoffrey Burn, and Simon Peyton Jones. A safe approach to parallel combinator reduction. *Theoretical Computer Science*, 56(1):17–36, January 1988.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[HP79] Matthew Hennessy and Gordon Plotkin. Full abstraction for a simple parallel programming language. In J. Bečvář, editor, *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science*, 1979, 108–120, Lecture Notes in Computer Science 74, Springer-Verlag.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[LGH*78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10:1–26, 1978.

[LP87] Peter Lee and Uwe Pleban. A realistic compiler generator based on high-level semantics. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, München, West Germany, January 21–23, 1987, 284–295, ACM.

[Mey87] Bertrand Meyer. Eiffel: programming for reusability and extendibility. *ACM SIGPLAN Notices*, 22(2):85–99, February 1987.

[Mos76] Peter D. Mosses. Compiler generation using denotational semantics. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science*, 1976, 436–441, Lecture Notes in Computer Science 45, Springer-Verlag.

[MW86a] Peter D. Mosses and David A. Watt. *Pascal: action semantics — towards a denotational description of ISO Standard Pascal using abstract semantic algebras.* Draft Version 0.3, Aarhus University, Computer Science Department, Aarhus, Denmark, September 5, 1986.

[MW86b] Peter D. Mosses and David A. Watt. *The use of action semantics.* Report DAIMI PB-217, Aarhus University, Computer Science Department, Aarhus, Denmark, August 1986. Appeared in [Wir86].

[Pag81] Frank G. Pagan. *Formal Specification of Programming Languages.* Prentice-Hall, 1981.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Plo81] Gordon D. Plotkin. *A structural approach to operational semantics.* Report DAIMI FN-19, Aarhus University, Computer Science Department, Aarhus, Denmark, September 1981.

[Plo83] Gordon D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II*, 1983, 199–223, North-Holland.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, September 1976.

[Spe82] David Spector. Ambiguities and insecurities in Modula-2. *ACM SIGPLAN Notices*, 17(8):43–51, August 1982.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[Tur85] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, 1985, 1–16, Lecture Notes in Computer Science 201, Springer-Verlag.

[Vaa86] Frits W. Vaandrager. *Process algebra semantics for POOL.* Report CS-R8629, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, August 1986.

[Weg72] Peter Wegner. The Vienna Definition Language. *ACM Computing Surveys*, 4(1):5–63, 1972.

[Wir86] M. Wirsing, editor. *Formal Description of Programming Concepts III: Proceedings of the Third IFIP WG 2.2 Working Conference*, Gl. Avernæs, Ebberup, Denmark, August 25–28, 1986, North-Holland.

[WSH77] J. Welsh, W. J. Sneeringer, and C. A. R. Hoare. Ambiguities and insecurities in Pascal. *Software—Practice and Experience*, 7, 1977.

A Note on Disjoint Parallelism

Krzysztof R. Apt
Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands
and
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712–1188
U.S.A.

Ernst-Rüdiger Olderog
Department of Computer Science
Christian-Albrechts University
2300 Kiel 1, Olshausenstrasse 40
West Germany

Abstract: We prove determinism of disjoint parallel programs by applying simple results on abstract reduction systems.

In this note we consider disjoint parallel programs as defined in Hoare [1972]. We assume from the reader the knowledge of **while**-programs (see e.g. De Bakker [1980]) and their semantics defined by means of transitions (see Hennessy and Plotkin [1979]). Two **while**-programs S_1 and S_2 are called *disjoint* if none of them can change the variables accessed by the other one, i.e. if

$$change(S_1) \cap var(S_2) = var(S_1) \cap change(S_2) = \emptyset,$$

where $change(S)$ is the set of variables of S which can be modified by it, i.e. to which a value is assigned within S by means of an assignment. Note that disjoint programs are allowed to read the same variables. For example, the programs

$$x := z \text{ and } y := z$$

are disjoint because $change(x := z) = \{x\}$, $var(y := z) = \{y, z\}$ and $var(x := z) = \{x, z\}, change(y := z) = \{y\}$.

On the other hand the programs $x := z$ and $y := x$ are not disjoint because $x \in change(x := z) \cap var(y := x)$.

Disjoint parallel programs are generated by the same clauses as those defining **while**-programs together with the following clause for *parallel composition*:

$$S ::= [S_1 \| \ldots \| S_n]$$

where for $n \geq 1$, $S_1, \ldots, S_n$ are pairwise disjoint **while**-programs. Thus we do not allow nested parallelism, but we allow parallelism to occur within sequential composition, conditional statements and **while**-loops.

Intuitively, a disjoint parallel program of the form $S \equiv [S_1 \| \ldots \| S_n]$ terminates if and only if all of its components $S_1, \ldots, S_n$ terminate; the final state is then the composition of the final states of $S_1, \ldots, S_n$.

Following Hennessy and Plotkin [1979] we define the semantics of disjoint parallel programs in terms of transitions. Intuitively, a disjoint parallel program $[S_1 \| \ldots \| S_n]$ performs a transition if one of its component performs a transition. Formally, we expand the transition system for **while**-programs by the following rule:

$$\frac{< S_i, \sigma > \; \to \; < T_i, \tau >}{< [S_1 \| \ldots \| S_i \| \ldots \| S_n], \sigma > \; \to \; < [S_1 \| \ldots \| T_i \| \ldots \| S_n], \tau >}$$

where $1 \leq i \leq n$.

Recall that computations of disjoint parallel programs are defined as those of **while**-programs. For example

$$\begin{array}{ll} & < [x := 1 \| y := 2 \| z := 3], \sigma > \\ \to & < [E \| y := 2 \| z := 3], \sigma[1/x] > \\ \to & < [E \| E \| z := 3], \sigma[1/x][2/y] > \\ \to & < [E \| E \| E], \sigma[1/x][2/y][3/z] > \end{array}$$

is a computation of $[x := 1 \| y := 2 \| z := 3]$ starting in σ.

Here E stands for the empty program and its occurrence denotes termination of the appropriate component. For example, $[E \| y := 2 \| z := 3]$ denotes a parallel program where the first component has terminated. As explained above a parallel program terminates if and only if all its components terminate. Consequently we identify

$$[E \| \ldots \| E] \equiv E.$$

Thus the final configuration in the above computation is the terminating configuration

$$< E, \sigma[1/x][2/y][3/z] > .$$

We define the partial and total correctness semantics $\mathcal{M}$ and $\mathcal{M}_{tot}$ of disjoint parallel programs by putting for a state σ

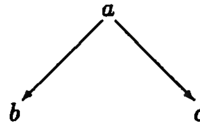
$$\mathcal{M}[[S]](\sigma) = \{\tau \mid < S, \sigma > \rightarrow^* < E, \tau >\}$$

where $\rightarrow^*$ denotes the transitive reflexive closure of $\rightarrow$, and

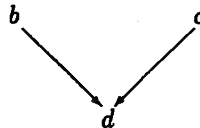
$$\begin{array}{rcl} \mathcal{M}_{tot}[[S]](\sigma) & = & \mathcal{M}[[S]](\sigma) \\ & \cup & \{\perp \mid S \text{ can diverge from } \sigma\}. \end{array}$$

Our aim is to prove that for a disjoint parallel program only one outcome for a given initial state is possible. In other words, for any disjoint parallel program S and state σ, $\mathcal{M}_{tot}[[S]](\sigma)$ has exactly one element. To this end, we need some results concerning abstract reduction systems.

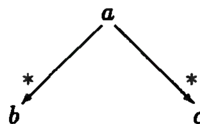
Definition 1 Let $\rightarrow$ be a non-empty binary relation. Denote by $\rightarrow^*$ the transitive reflexive closure of $\rightarrow$. $\rightarrow$ satisfies the *diamond property* if for all a, b, c such that $b \neq c$



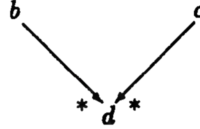
implies that for some d



$\rightarrow$ is called *confluent* if for all a, b, c



implies that for some d



□

The following lemma due to Newman [1942] is of importance to us.

Lemma 2 (Newman's Lemma) If a relation $\rightarrow$ satisfies the diamond property then it is confluent.

Proof. Suppose that $\rightarrow$ satisfies the diamond property. Let $\rightarrow^n$ stand for the n-fold composition of $\rightarrow$. A straightforward proof by induction on $n \geq 0$ shows that $a \rightarrow b$ and $a \rightarrow^n c$ implies that for some $i \leq n$ and some d, $b \rightarrow^i d$ and $c \rightarrow^\epsilon d$. Here $c \rightarrow^\epsilon d$ iff $c \rightarrow d$ or $c = d$. Thus $a \rightarrow b$ and $a \rightarrow^* c$ implies that for some d, $b \rightarrow^* d$ and $c \rightarrow^* d$.

This implies by induction on $n \geq 0$ that if $a \rightarrow^* b$ and $a \rightarrow^n c$ then for some d, $b \rightarrow^* d$ and $c \rightarrow^* d$. This proves confluence. □

We shall also need the following lemma.

Lemma 3 Suppose $\rightarrow$ satisfies the diamond property and that $a \rightarrow b$, $a \rightarrow c$, $b \neq c$. If there exists an infinite sequence $a \rightarrow b \rightarrow \ldots$, then there exists an infinite sequence $a \rightarrow c \rightarrow \ldots$.

Proof. Consider an infinite sequence $a_0 \rightarrow a_1 \rightarrow \ldots$ where $a_0 = a$ and $a_1 = b$.
Case 1. For some $i \geq 0$, $c \rightarrow^* a_i$.

Then $a \rightarrow c \rightarrow^* a_i \rightarrow \ldots$ is the desired sequence.

Case 2. For no $i \geq 0$, $c \rightarrow^* a_i$.

By induction on i we construct an infinite sequence $c_0 \rightarrow c_1 \rightarrow \ldots$ such that $c_0 = c$ and for all $i \geq 0$, $a_i \rightarrow c_i$. For $i = 0$, c_i is already correctly defined.

Consider now the induction step. We have $a_i \rightarrow a_{i+1}$ and $a_i \rightarrow c_i$ for some $i > 0$. Also, since $c \rightarrow^* c_i$, by the assumption $c_i \neq a_{i+1}$. Again by the diamond property for some c_{i+1}, $a_{i+1} \rightarrow c_{i+1}$ and $c_i \rightarrow c_{i+1}$. □

Define now for an element a in the domain of $\rightarrow$

$$\begin{array}{rcl} yield(a) & = & \{b \mid a \rightarrow^* b, b \text{ is } \rightarrow\text{-maximal}\} \\ & \cup & \{\perp \mid \text{ there exists an infinite sequence } a \rightarrow a_1 \rightarrow \ldots\} \end{array}$$

where b is called $\rightarrow$-maximal if for no c, $b \rightarrow c$.

Lemma 4 Suppose that $\rightarrow$ satisfies the diamond property. Then for every a, $yield(a)$ has exactly one element.

Proof. Suppose that for some $\rightarrow$-maximal b and c, $a \rightarrow^* b$ and $a \rightarrow^* c$. By Newman's Lemma for some d, $b \rightarrow^* d$ and $c \rightarrow^* d$. By the $\rightarrow$-maximality of b and c, both $b = d$ and $c = d$, i.e. $b = c$.

Thus the set $\{b \mid a \rightarrow^* b, b \text{ is } \rightarrow\text{-maximal}\}$ has at most one element. If it is empty, then $yield(a) = \{\perp\}$ and we are done.

Otherwise it has exactly one element, say b. Assume by contradiction that there exists an infinite sequence $a \rightarrow a_1 \rightarrow \ldots$. Consider a sequence $b_0 \rightarrow b_1 \rightarrow \ldots \rightarrow b_k$ where $b_0 = a$ and $b_k = b$. Then $k > 0$. Let $b_0 \rightarrow \ldots \rightarrow b_\ell$ be the longest prefix of $b_0 \rightarrow \ldots \rightarrow b_k$ which is an initial fragment of an infinite sequence $a \rightarrow c_1 \rightarrow \ldots$. Then ℓ is well defined, $b_\ell = c_\ell$ and $\ell < k$, since b_k is $\rightarrow$-maximal. Thus $b_\ell \rightarrow b_{\ell+1}$ and $b_\ell \rightarrow c_{\ell+1}$. By the definition of ℓ, $b_{\ell+1} \neq c_{\ell+1}$. By Lemma 3 there exists an infinite sequence $b_\ell \rightarrow b_{\ell+1} \rightarrow \ldots$. This contradicts the choice of ℓ. Thus $yield(a) = \{b\}$. □

We now wish to apply Lemma 4 to the case of disjoint parallel programs. To this purpose we prove first the following lemma.

Lemma 5 (Diamond Property). Let S be a disjoint parallel program and σ a state. Whenever

$$\begin{array}{c} < S, \sigma > \\ \swarrow \quad \searrow \\ < S_1, \sigma_1 > \neq < S_2, \sigma_2 >, \end{array}$$

then for some configuration $< T, \tau >$

$$\begin{array}{c} < S_1, \sigma_1 > \quad < S_2, \sigma_2 > \\ \searrow \quad \swarrow \\ < T, \tau >. \end{array}$$

Proof. By the format of the transition rules, S is of the form $[T_1 \| \ldots \| T_n]$ for some pairwise disjoint **while**-programs $T_1, \ldots, T_n$ and there exist **while**-programs T_i' and T_j', with $i \neq j$, $1 \leq i, j \leq n$ such that

$$\begin{array}{c} S_1 = [T_1 \| \ldots \| T_i' \| \ldots \| T_n], \\ S_2 = [T_1 \| \ldots \| T_j' \| \ldots \| T_n], \\ < T_i, \sigma > \rightarrow < T_i', \sigma_1 >, \\ < T_j, \sigma > \rightarrow < T_j', \sigma_2 >. \end{array}$$

Let

$$T = [T_1' \| \ldots \| T_n']$$

where for $k = 1, \ldots, n$ such that $k \neq i$ and $k \neq j$

$$T'_k = T_k$$

If $\sigma_1 \neq \sigma$ then the transition $< S, \sigma > \to < S_1, \sigma_1 >$ consists of executing an assignment statement and then $\sigma_1 = \sigma[d_1/u_1]$ for some variable u_1 and element d_1 of the domain D.

Similarly if $\sigma_2 \neq \sigma$ then $\sigma_2 = \sigma[d_2/u_2]$ for some variable u_2 and element d_2 of the underlying domain over which all variables range. Here $\sigma[d/u]$ stand for the state differing from σ only on the variable u to which it assigns the value d.

We now define τ depending on the cardinality of the set $\{\sigma, \sigma_1, \sigma_2\}$:

$$\tau = \begin{cases} \sigma & \text{if } card\{\sigma, \sigma_1, \sigma_2\} = 1, \\ \rho & \text{if } card\{\sigma, \sigma_1, \sigma_2\} = 2 \text{ and} \\ & \quad \{\sigma, \sigma_1, \sigma_2\} = \{\sigma, \rho\}, \\ \sigma[d_1/u_1][d_2/u_2] & \text{if } card\{\sigma, \sigma_1, \sigma_2\} = 3. \end{cases}$$

If $\tau = \sigma[d_1/u_1][d_2/u_2]$ then $< S_1, \sigma_1 > \to < T, \tau >$. But by the disjointness condition $\tau = \sigma[d_2/u_2][d_1/u_1]$ which proves that also $< S_2, \sigma_2 > \to < T, \tau >$.

Other cases are straightward and left to Jaco. □

As an immediate corollary we obtain the desired result.

Theorem 6 (Determinism) For every disjoint parallel program and a state σ, $\mathcal{M}_{tot}[[S]](\sigma)$ has exactly one element.

Proof. By Lemma's 4 and 5. □

References

[1] J.W. de Bakker [1980], Mathematical Theory of Program Correctness, Prentice-Hall, Englewood Cliffs, N.J., 1980.

[2] M.C.B. Hennessy and G.D. Plotkin [1979], Full abstraction for a simple programming language, in: Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 74 (J.Bečvar, ed.), pp.108-120, 1979.

[3] C. A. R. Hoare [1972], Towards a theory of parallel programming, in: Operating Systems Techniques (C.A.R. Hoare, R.H. Perrot, eds.), pp. 61–71, Academic Press, 1972.

[4] M.H.A. Newman [1942], On theories with a combinatorial definition of "equivalence", *Ann. Math, 43*, pp. 223–243, 1942.

PLAYING AT SEMANTICS

P.C.Baayen
C.W.I., Amsterdam

Who was it that said: our playing is learning, our work is play? Certainly, among logicians, computer scientists and mathematicians there have always been many who loved to devise puzzles, make up puns and play with words and meanings. Witness the department of Scientific American, first called (in the days of Martin Gardner) *Mathematical Games*, then (run by Douglas R. Hofstadter) *Metamagical Themas*, and now contributed by A.K.Dewdney as *Computer Recreations* .

A very creative and influential contributor to the genre was the Rev. Charles Lutwidge Dodgson, lecturer in mathematics in Oxford, and to most people only known under his pen-name Lewis Carroll. His Diversions and Digressions, the epic Hunting of the Snark, and in particular the two Adventures of Alice have always drawn the attention of mathematicians and logicians. For both for the Snark and the Alice books Martin Gardner prepared valuable annotations, not only providing background information and literary associations, but in particular drawing attention to the logical puzzles and plays hidden behind the fancyful stories.

Naturally, ideas and characters from Lewis Carroll have been used again and again by logicians and philosophers to illustrate fine points of their argument. And if you have to prepare an introductory speech, "Alice" is a rich mine of metaphores. For instance, if you want to make the point that computer scientists in general, and those studying concurrent processes in particular, have to work extremely hard to keep abreast the rapidly changing technology, then it might be enlightening to quote the Red Queen, from the second chapter in "Through the Looking Glass". After running with the queen as fast as she could for quite a while, Alice remarks on her observation that no progress has been made. "Well, in *our* country", said Alice, still panting a little, "you'd generally get to somewhere else - if you ran very fast for a long time as we've been doing." "A slow sort of country!" said the Queen. Now, *here*, you see, it takes all the running *you* can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that."

Several books dealing with logic, computability or artificial intelligence refer to Lewis Carroll's Oeuvre. For instance, Douglas Hofstadter in "Gödel, Escher, Bach" has Carrollian creatures taking part in the dialogues. And the logician Raymond Smullyan even wrote a whole book around Alice, called "Alice in Puzzle-land" (New York, 1982). A prime example of *pure* logic, if I may follow the classification of Craig Smoryński.

Maybe this needs some explaining. Smullyan - author of outstanding works on theoretical logic, such as "Theory of Formal Systems" (Princeton, 1961) and "First-Order Logic" (Berlin &c, 1968) - has enriched the literature with a number of books in which complex logical results are treated and explained through the device of logical puzzles. Apart from "Alice in Puzzle-land", there are "What is the Name of this Book" (1978), "The Lady and the Tiger" (1982), "To mock the Mocking Bird" (1985) and "Forever Undecided. A Puzzle Guide to Gödel" (and possibly others). Not only the Gödel theorems are explained and elucidated by means of puzzle sequences, but also aspects of combinatory logic and of proof theory. "Forever Undecided" was reviewed by Smoryński in the February 1989 issue of the American Mathematical Monthly; in this review, Smoryński submits that "the difference between pure and applied logic is merely one of where the application lies - in the sacred world of mathematics or the profane world of everyday affairs". Now it may seem as if Smullyan's puzzles deal with everyday affairs, like a defense lawyer jumping up and shouting "That's not true" when his client is accused that, *if* he did the terrible deed, he must have had an accomplice (thus exhibiting a defective grasp of material implication); or such as islands full of knights (who are unable to lie) and knaves (who are unable to speak the truth), or Humpty Dumpty confronting Alice with complex variations on the Barber Paradox. One is tempted to conclude, therefore, that they are applied logic. However, as far as Smullyan is concerned, Smoryński arrives at a different conclusion: "This is pure logic, not applied. It falls short of an application to epistemology because of the imaginary nature of the island and its oversimple picture of human nature".

As a true example of applied logic, Smoryński gives us an island in the Aleutes, inhabited (exclusively) by Russians and Americans, some of which are patriots and some are traitors. As it happens, patriots always and only speak the truth to their compatriots, while traitors always and only speak the truth to those of the opposite nationality. If somebody - either an American or a Russian - asks a native "Are you an American or a Russian?", he will not be able (whatever the answer) to conclude anything concerning the nationality of the native, but he will be able to reason out whether his respondent is a traitor or a patriot. The logical analysis of how Americans or Russians reason in solving this puzzle is, according to Smoryński, an instance of *applied* logic. (Amazingly, it turns out - as everyone can easily check for himself - that Russians and Americans reach different conclusions.)

The example is enlightening. It is surely to be hoped that Smoryński will fulfill his promise and indeed write a monograph on "Beyond Detente; A Puzzle Guide to Geo-Political Realities". But of greater importance is the application of Smoryński's classification to the scientific work of Jaco de Bakker and his school. Indeed, isn't it obvious that the semantics of programming languages, particularly so if dealing with parallel object-oriented approaches, is so far removed from "the profane world of everyday affairs" that evidently it is to be classified as *pure* logic! It is not clear to me whether this conclusion will please Jaco; after all, he has been singularly succesful in convincing pragmatic, application-oriented people to pay for his research. But in any case this explains the great attraction to logicians of Jaco de Bakker and his department.

The Carrollian universe has not only been a source of ideas to logicians and their ilk: it has also inspired authors with more literary aspirations. One of those, Gilbert Adair, has given us a third Alice adventure: "Alice through the Needle's Eye" (Macmillan, London &c, 1984). It is a pleasant book to read, almost as full of puns and playings on words and meanings as the real Adventures. Where it falls short, is in the logic tricks and puzzles.

Alice, falling through the eye of the needle she is vainly trying to thread, drops down to a country full of Carrollian creatures. Enjoying her birds-eye view, she considers herself to be the first orni-theologist. After a safe landing in an A-stack (inhibited by a Cockney mouse who is convinced she is 'Alley's comet), Alice meets many interesting creatures: camels of course (after all she seems to remember that it is in the nature of camels to go through the eye of a needle), an anony-mouse who wrote all those poems in anthologies signed Anon., and the otter and hamster who together built a dam and then could not agree whether to call it *Otterdam* or *Hamsterdam*. At one stage Alice finds herself in a restaurant, with the waiter a nicely dressed Frog (the illustrations by Jenny Thorne, by the way, do convincingly look like they might have been drawn by Tenniel). Alice asks him what dish he recommends. "We-ll", replied the Frog in a slow drawl, "what I *can't* recommend is the Frog's Legs, you know. Order those - and you'll see as how the service slows down something dreadful".

One happening is an Election, with an Emu running for office. This is one of the episodes where a chance for a logical joke is missed. Alice askes his Emu-nence what election promises he makes. "The Emu gave her such a piercing stare, that any other little girl of her age would have been quite put out by it. "Ahem - what promises do I make? Why, I promise - I promise - I promise *not* to break my promise - and that's a promise! Next question!". "

Certainly, Lewis Carroll would have handled this scene differently. Why, the Emu's answer is logically speaking without content (and that probably is the sole intention of the author); but Alice (and we) would have had more food for thought if the Emu had said something like "you annoy me so much - I promise you I'll break this very promise". Or wouldn't it be more fun to have a politician say: "I promise this is the last promise I'll ever break"? It might be nice if Jaco, or some of the people around him that are well-versed in temporal logic, would sit down and make up really clever promises; if somebody does, please tell me what comes out of it!

One of the denizens of the country behind the needle's eye deserves special mention: the Grampus. He is an impressive character; one just can't get around him or ignore his presence. From his appearance and behaviour (he even has the ability to read and use *italics*) it is immediately clear that he must be a learned scientist; this also appears from some of his idiosyncrasies (he regularly lectures to Alice, and he is so absent-minded that he sometimes even forgets that he is absent-minded, and remembers everything). In fact, he is Headmaster in the School of Whales. Although he is (on first sight) a ferocious man-eater, he turns out to be quite amiable and friendly, once you learn to know him better. And on top of that, he is a specialist on semantics. Allow me to quote from the Grampus lecturing to Alice.

"Meaning, my dear, is a rare and precious substance", the Grampus gravely replied: "so precious that, if my opinion were asked about it, it should

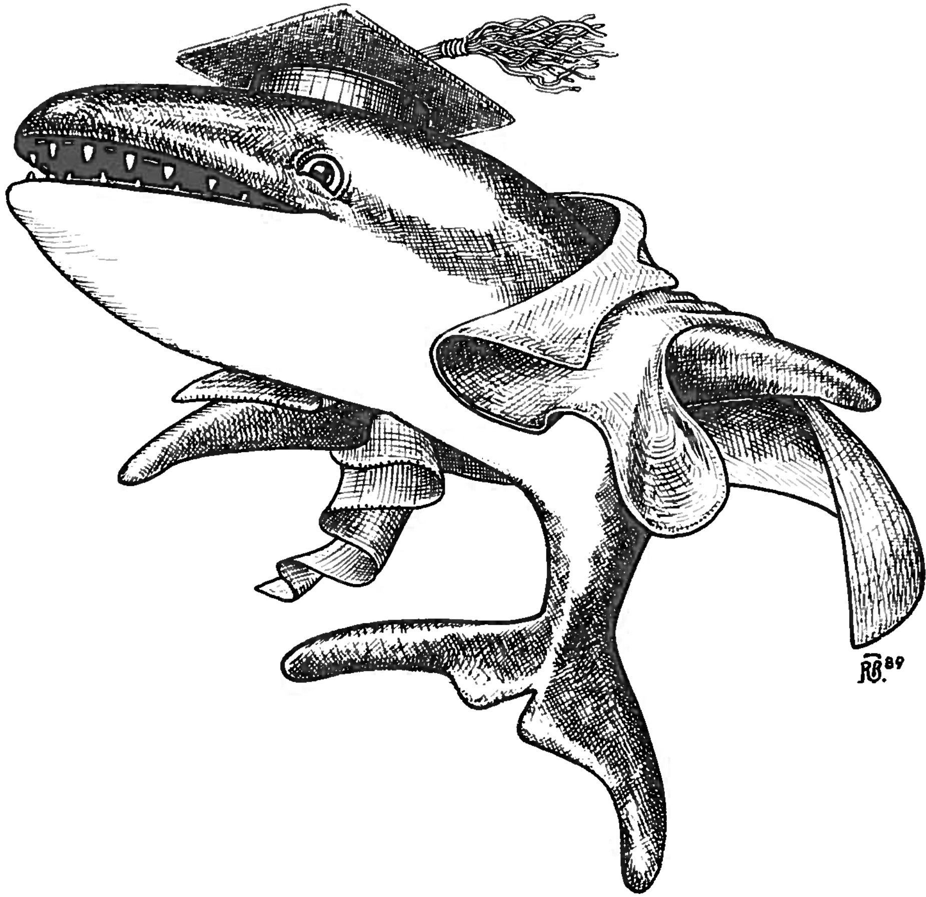
be preserved under a bell jar in the Museum - on view to the Public, Tuesdays and Fridays, at sixpence a time".

"But dictionaries are full of meanings", objected Alice, who remembered consulting one not so long ago.

"Full of *meanings*, perhaps, but empty of *meaning*", said the Grampus. "As I ought to know, for I tried reading a dictionary once - just as a change from my Auto-biography, you understand - but the story in it was the dullest and most meaningless I ever read. And the reason for that is, that the best meanings can't ever be written down, that's how precious they are."

The Grampus, to me, somehow, however imperfectly, combines several features which I admire in Jaco de Bakker. For that reason, I would have liked to reproduce here one of his portraits by Jenny Thorne. Alas, it was too late to obtain permission from artist and publisher. Therefore I decided to go one better, and I asked Tobias Baanders to give me his artistic impression of a "Headmaster of the School of Whales". As always, Tobias came up with a beautiful drawing. I am pleased to dedicate it here to

Jaco de Bakker.



On formal and informal reasoning in program construction

R.J.R. Back, Åbo Akademi

To Jaco

The problems with constructing and maintaining large programs are well known. The low quality of software is certainly one of the main stumbling factors in the computerization process, and one that has caused more frustration than any other aspect of this technology. Although the problem is so well recognized, it seems very difficult to cure. Different proposals have been put forward, such as higher level specification and programming languages, a better organization of the program construction process and quality control measures, improved testing methods and more formal methods for program construction. It is this last proposal that we will study in more detail in this short note.

Before we look at the formal methods, to see whether they promise a solution to the software quality problem, let us turn the question upside down. Rather than ask why the present informal program construction methods do not work well enough, we ask why they do work as well as they do. The programs may be bugridden and difficult to maintain, but for most cases they do work. There must be something in the present informal methods that does go quite a long way towards the goal of quality software. This suggests that the informal methods are good in themselves, but that they are not sufficient for achieving very high levels of quality.

The approach we want to put forward here is that the informal and the formal approaches complement each other, in a way that is in close analogue to the usual way in which everyday science is conducted. Our starting point is the view of scientific work as it is done in e.g. the natural sciences, where empirical and theoretical studies form a close symbiosis. This view can be described roughly (and somewhat naively) as follows.

1. Understanding a new phenomenon in e.g. physics starts by empirical observations. The phenomenon is identified and its manifestations are studied by empirical experiments. These experiments are carried out until some kind of general picture emerges, which provides some (often partial) explanation of what is going on. The experiments are done against a body of prior knowledge, both theoretical and experimental, which guides the experiments. In other words, the experiments are not done in a vacum, but experimenters have a reasonably good idea of what they are looking for.

2. Based on the initial experiments, an explanation of the phenomenon is attempted. This is a generalization, i.e. a model that abstracts away from the specific cases that have been studied in the experiments to provide a more general description of the phenomenon. The model also predicts the outcome of experiments which have not yet been made.

3. This general explanation is then tested against the reality by performing new experiments. The outcome of these experiments is predicted by the generalization. The new experiments should be chosen such that the possibility of falsifying the model is as big as possible. In this way erroneous generalizations can be dispensed with quickly.

4. The new experiments often do not totally falsify the generalization, but point at weaknesses in it. Based on these new experiments, the general explanation is modified so that the new experiments are also taken into account. The modified explanation is then again tested experimentally. The cycle is repeated until there is no contradicting evidence from experiments. The testing of the hypothesis does not have to be made by the same researchers, it may be done by different research groups, and the activity may be stretched over space and time (even over very long periods, like years, decades and centuries).

Different philosophical theories of science differ in how this enquiry is carried out: when a hypothesis or theory can be considered confirmed, how the experiment should be tested and so on. When certain assumptions about the physical reality are satisfied, these questions can e.g. be analyzed by statistical means. The theories agree, however, on the need for the duality between theory and experiments, and on the different nature of these two activities. Theory is *deductive* while experimentation is *inductive*. A theory should be internally consistent, and it should be built according to standard mathematical requirements of rigour and exactness: the definitions should be exact and clear, the theorems should be well defined and the proofs should be rigourous. The theories must have an interpretation in the real world to be useful, but they must also by themselves form a consistent and rigorously developed mathematical theory.

Experimentation studies the phenomenon as it manifests itself in reality. It is often carried out against a body of theory, so it is usually aimed at testing some hypothesis by which the theory is extended, but not all of it. Experimentation seems to have two different purposes: on one hand it is used as an inspiration for formulating new theories and explanations (hypothesis), on the other hand it is used to test the validity of existing hypothesis, formed on the basis of previous experiments or derived (deductively) from hypothesis formed on the basis of previous tests.

Informal methods in program construction Let us now consider how the above relates to program construction. Our first observation is that informal program construction, as it is done in practice, works because it applies the experimental-deductive scientific method. In constructing a program, one thinks in terms of specific examples (test cases). These examples are generalized to a program. The program is a true generalization, because it is intended to also apply to other cases than those considered explicitly. After constructing the program, it is tested on a collection of test cases. These form the new experiments. Usuallly the test cases are constructed so that the likelyhood of detecting an error is as big as possible, i.e. the purpose is to refute the hypothesis that the program works correctly. If a test reveals an error, the source of the error is located (a bug), the hypothesis is revised (the bug is fixed) and testing of the new hypothesis is started. This process is repeated until no further errors are detected, in which case the program is accepted as correct (delivered to the customer), i.e. the hypothesis (or theory) is considered to be confirmed.

Program construction is thus a classical example of the experimental scientific method: the interplay between experiments and theory. Even the criteria for accepting a hypothesis are the same, the hypothesis is accepted when all experiments confirm it. Also, this acceptance is conditional, as it is recognized that further experimentation may still invalidate the hypothesis (as it usually does). The informal program construction method does indeed work in practice, for the same reason that the experimental scientific method works in general: there is enough regularity in the phenomen studied that a reasonably careful testing gives a good basis for assuming that the theory is valid for most cases that can occur in practice.

Formal methods in program construction After having convinced ourselves that the practical approach to program construction rests on good, solid scientific methods, let us now

turn to the issue of formal reasoning. The main question here is whether we can do better than this. Our first observation is that the analogy between computer programs and the physical reality that natural scientists study is not complete: the physical reality is given and we only know about it through our experiments, whereas the program is constructed by ourselves. In other words, we have complete information about the program and its working, whereas we do not have complete information about the physical reality. Taking the experimental scientists approach to program construction therefore ignores information that is available. The program that we ourselves have constructed is treated as a black box, the inner workings of which is unknown to us. Or it is treated as a white (or transparent) box, whose inner workings can be seen, but where the function is so complex that it cannot be understood sufficiently to be able to predict the outcome in general.

The analogy we are looking for is maybe then not the way in which the physicist construct theories, but more the way in which mathematicians construct theories, because the latter are in the same situation as we are: they are working in a reality that they have constructed themselves. The problems in mathematics come from a similar source as in programming: the reality that has been constructed is so complex that its working cannot be understood in full detail at once, even if it is in fact known in complete detail.

The way in which mathematicians prove theorems might be a good starting point. We can consider a program as a theorem: it asserts that a certain algorithmically defined function satisfies certain desirable properties. In mathematics we have an informal and a formal way of reasoning. The informal way consists in constructing special cases or examples which illustrate the phenomenon being considered. From these examples a generalization is then made, which again covers more cases and examples than what has actually been considered. These generalizations (lemmas, definitions etc.) are then used in the proof. The proof itself proceeds by formal reasoning. If the theorem can be shown to be correct by formal (or mathematically rigorous) reasoning, then it is accepted as correct. If, however, the proof cannot be constructed, then the usual recourse is to analyze the difficulty by trying to construct new examples and cases where the difficulty is identified. Based on this experimental study, the generalizations are modified so that the difficulties are avoided. This process may be continued until the proof succeeds or until a counterexample is found which finally kills all hopes that the theorem could be fixed to be true.

Comparing this approach to the purely experimental approach by the physicists, we see one major difference. The confirmation of the hypothesis is not done by testing but by deductive reasoning. In other words, the hypothesis is accepted if it can be proved rigorously that it is correct. Only if the proof does not succeed is the experimental stage re-entered. Hence, the role of experiments and testing in mathematics is not to confirm a hypothesis but to generate new hypothesis or to refute the given hypothesis by counterexamples. Still, the experimental part of the mathematical work is very important, and often provides the inspiration and the real workbench where mathematical discoveries are made.

A view of program construction How would we apply this same methodology in the construction of computer programs. First of all, we need to recognize the important role that experimentation plays in program construction, also in the case where the goal is to construct programs that are formally verified correct. In the same way as in mathematics, it provides the playground on which inventions are made. These inventions are based on intuition, i.e. based on the understanding of the problem that has been gained from analyzing a collection of examples, in combination whith the general understanding that has been gained previously. In part it can also arize from purely formal manipulations, based on the mathematical structure of the hypothesis and the formal deductions that can be made from it. However, the first source

of inventions would seem to be the more important one, at least as long as the mathematical theory of program construction is as rudimentary as it is today, with few powerful theorems and results to rely upon and no uniform basic theory agreed upon.

The second issue is to recognize the role of verification in program construction: It is the only tool available for confirming that the program works correctly. Difficult as it may be, we cannot hope for achieving greater confidence in the correctness of our programs unless we can verify mathematically that they are correct and have some kind of machine checking of the proofs.

Program construction thus takes place in two parallel but separate worlds, the *experimental testing world* and the *deductive confirmation world*. The interplay between these two worlds occurs in most program construction situations, from direct verification that a program meets its specification to program construction by stepwise refinement and the construction of precise specifications for some required sofware product. To only have experimentation and restrict oneself to experimental confirmation of program correctness is to settle for less than is achievable. To only accept formal reasoning is to deny the main source of inspiration and invention that underlies the construction of efficient and useful programs.

The restriction to experimentation alone is common and even prevailing today, and is usually based on the view that program verification does not work in practice anyway, at least not for larger programs (an issue that cannot be considered resolved yet). The restriction to formal methods only has not to my knowledge really been put forward seriously.A somewhat less extreme but still one sided position is, however, to accept that the informal testing/experimentation activity is needed, but that it need not be recorded in the development process: all experimentation should be done behind the scenes and only the formal reasoning should be recorded. This position, analogues to Gauss' view on mathematical proofs, leads to terse and unintuitive presentations of program constructions. (Often this is only a question of how the material is presented and not an explicitly taken standpoint).

Not recording the intuition and experimentation behind the construction makes the the programs, proofs and program derivations difficult to follow and understand. In this one follows established mathematical tradition: the definitions and proofs are often presented without explaining the intuition behind them. This makes it difficult for people who try to get into a subject to understand what is being done and why. The analogy with poorly documented program text should be obvious here. The need for giving the intuition is felt most strongly in textbooks in mathematics and, as a consequence, the purely mathematical treatment is extended with intuitive explanations, examples and counterexamples as well as exercises. All these serve the same purpose, to present the experimental background that has lead to the formulation of the theory.

A consequence of not recognizing the important role of experimentation is that the need for methods and tools which support the experimentation phase of formal program construction is also not recognized. Consequently no tools are built for this. Such tools can be constructed today; the modern workstation environment with its powerful graphic tools should provide a good workbench for experimentation. The important aspects of such an environment is that it should be very flexible and strongly visually oriented. It should encourage both informal experimentation and formal deduction and verification. It should also record both these activities in a consistent and structured way, so that the necessary documentation is created automatically during program construction, as a byproduct of it.

An Algebra for Process Creation

Jos C.M. Baeten

Frits W. Vaandrager

Department of Software Technology,
Centre for Mathematics and Computer Science,
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

In this paper, we study the issue of process creation from an algebraic perspective. The key to our approach, which is inspired by the work of AMERICA & DE BAKKER [AB], consists of giving a new interpretation to the operator symbol · (sequential composition) in the axiom system BPA of BERGSTRA & KLOP [BK1,2,3]. We present a number of other models for BPA and show how the new interpretation of · naturally generalises the usual interpretation in BPA or ACP. We give an operational semantics based on Plotkin style inductive rules, and give a complete finite axiomatisation of the associated bisimulation model.

1. INTRODUCTION.

In process algebra theories like CCS, CSP, MEIJE and ACP, not much attention has been paid so far, to the concept of process creation. Instead, parallel composition is used as a primitive constructor of concurrent systems. In SMOLKA & STROM [SS] and VAANDRAGER [VA], process algebra semantics is given for languages with process creation (NIL resp. POOL), but there the process creation construct is translated to an architectural expression with parallel composition.

A first attempt to deal more directly with process creation in an algebraic setting is described in BERGSTRA [B], where the axiomatic system ACP is extended with a mechanism for process creation. The key axiom here is

$$E_\phi(cr(d)\cdot x) = \overline{cr}(d)\cdot E_\phi(\phi(d) \| x).$$

The operator E_ϕ denotes an environment in which process creation can take place. If an action $cr(d)$ is performed in this environment, a process $\phi(d)$ is created and placed in parallel with the remaining process.

Since process creation is an important concept, present for instance in ADA, NIL, POOL and UNIX, it seems worthwhile to look for a more direct and compositional treatment of process creation which does not need a global environment like an E_ϕ-operator. Here, we profit to a large extent of the work of AMERICA & DE BAKKER [AB]. The simple but crucial observation which they make, is that in order to give a compositional semantics to process creation, one has to interpret the sequential composition differently. As an example consider the expression

$$a\cdot new(b\cdot c)\cdot d.$$

Both authors are sponsored by ESPRIT project 432, METEOR (A formal integrated approach to industrial software development), and RACE project 1046, SPECS (Specification and Programming Environment for Communication Software).

The intuitive semantics of this expression is a process which first performs a, after which a new process is created doing b followed by c. The newly created process executes in parallel with the continuation d. Thus, the traces of this process are abcd, abdc and adbc. Consequently, we cannot interpret $x \cdot y$ as 'first do x and then y' (as is usual), because in a setting with process creation process x may continue after process y has started.

For this reason, in AMERICA & DE BAKKER [AB], a new semantical operator : is introduced, which serves as the interpretation of · in a setting with process creation. In the algebra for process creation that we present in this paper, we will interpret the · as a *continuation* operator in essentially the same way as in [AB]. But before we come to this operator, we first give an extensive overview of a number of other interpretations of ·. What all these interpretations have in common with the continuation operator, is that in a setting with alternative composition (+), they all satisfy the axioms of BPA (Basic Process Algebra) of BERGSTRA & KLOP [BK1,2, 3]:

$$x + y = y + x \qquad (x + y) \cdot z = x \cdot z + y \cdot z$$
$$(x + y) + z = x + (y + z) \qquad (x \cdot y) \cdot z = x \cdot (y \cdot z)$$
$$x + x = x.$$

Most of the discussion of this paper takes place in the setting of interleaving semantics. However, we show that a particular interpretation of · as *sequential composition* (like in ACP) and also our interpretation of · as continuation, can both be lifted in a natural way to the world of *event structures* of WINSKEL [W]. In both these interpretations, we have an instance of *action refinement* in the sense of [CDP] and [GG]. In fact, and this is surprising, sequential composition and continuation have the *same* definition on event structures, only sequential composition is defined on a more restricted domain of processes. Hence the rather substantial differences between the two operators on the level of interleaving semantics almost disappear on the level of True concurrency.

Whereas in AMERICA & DE BAKKER [AB] operational as well as denotational models are presented (and proven to be equivalent), we concentrate on operational models in this paper. As is done in [AB], we use Plotkin-style rules for the operational semantics. There are a number of differences, however.

First, we want all rules to be as simple as possible, and each rule should embody a clear intuition about a certain operator. Therefore, we reject a rule like

$$\langle \ldots, (s_1;s_2);r, \ldots, w\rangle \rightarrow \langle \ldots, s_1;(s_2;r), \ldots, w\rangle,$$

which occurs in [AB]: we think it is not part of a natural operational intuition about the ;-operator that brackets can move to the right.

A second design criterion that we used in the construction of our operational semantics is that all rules should be in the *tyft/tyxt* format of GROOTE & VAANDRAGER [GV]. This format poses certain restrictions on the inductive rules which guarantee that bisimulation equivalence is a congruence. Thus, any set of rules in *tyft/tyxt* format immediately induces an abstract compositional semantics. In [GV] it is shown that this format cannot be generalised in any obvious way, unless one is willing to work in a setting of terms over a many sorted signature, or use rules with negative hypotheses.

Our third design criterion was that the transition systems generated by the inductive rules should contain no silent or internal steps. If such transitions are present, one is more or less forced to say something about the nature of τ and to choose whether one adopts all of Milner's τ-laws or only a few of them. We prefer to separate the issue of abstraction from other concerns.

A final design criterion is upward compatibility with a non-interleaved event structure semantics. By now, many algebraic concurrency languages have been provided with a non-interleaved semantics (see e.g. [DDM], [O], [BC], [W]). We think that a general requirement for an interleaved semantics of a concurrent language is that there exists a natural non-interleaved semantics which is compatible with it. More specifically, we require that there is an event structure semantics with this property. The idea is that event structures (see WINSKEL [W]) constitute one of the most important domains for 'True' concurrency, and one must have a good reason to present a semantics which is incompatible with them. We show how some proposals for an operational semantics can be discarded because it is unclear how they could meet this last requirement.

In section 3, we present operational rules for a simple language APC for concurrent communicating processes with process creation. We claim that these rules meet all the requirements above. An interesting feature of these rules is that one of them has a look-ahead of more than one action: in order to compute the initial transitions of process $x \cdot y$, one needs information about the first *two* transitions of x. This implies in particular, that our $\cdot$ operator is not definable in terms of CCS, CSP, MEIJE or ACP.

In section 4, we present a sound and complete axiomatisation of the bisimulation semantics induced by the rules for APC. This axiomatisation uses a number of auxiliary operators.

With a number of examples, we illustrate in section 5 how APC can be used to specify concurrent systems, and how identities between processes can be proved algebraically.

2. BASIC PROCESS ALGEBRA.

2.1 The aim of this paper is to give an algebraic treatment of the feature of process creation. It will turn out that the key to our solution consists of giving a new interpretation to the operator symbol $\cdot$ in the axiom system BPA (Basic Process Algebra) of BERGSTRA & KLOP [BK2,3]. Therefore, we start with a review of BPA. We will see that there exist at least five very different interpretations of the operator symbol $\cdot$. One thing that all these interpretations have in common is that the laws of BPA are satisfied, and this similarity may be considered as a surprising fact.

2.2 BPA starts from a given set A of **actions**. These actions, denoted by a,b,c,... are constants in the language. Further, BPA has two binary operators: **sum**, denoted +, and **product**, denoted $\cdot$. Processes x,y,... constructed with these operators will always satisfy the axioms in the following equational specification BPA.

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5

TABLE 1. BPA.

Of all operators, · will always bind the strongest, and + the weakest. Thus, $x \cdot y + z$ means $(x \cdot y) + z$. We often write xy instead of $x \cdot y$. We denote the set of closed terms over BPA by $\mathbb{T}(BPA)$.

2.3 In the work of BERGSTRA & KLOP [BK2,3], the elements of A are often called **atomic actions**, the + operator is called **alternative composition**, and the · operator is called **sequential composition**. The intuition is that actions a,b,... are events without positive duration in time; they are atomic and instantaneous. The interpretation of $(a + b) \cdot c$ is a process that first chooses between executing a or b and, second, performs the action c after which it is finished. Since time has a direction, product is not commutative; but sum is, and in fact it is stipulated that the options possible in each state always form a *set* (axioms A1, A2, A3). The other distributive law $x(y + z) = xy + xz$ is not included, because the moment of choice between y and z in the two processes is different.

We would like to stress again that this is just one possible interpretation of the elements of the signature of BPA; we will consider other interpretations in the sequel.

2.4 SEQUENCING.
We will now present our first model for BPA. Like all models in this paper, it is defined using **structural rules** in the style of PLOTKIN [PL]. We introduce, for each constant $a \in A$, a binary **action relation** $\xrightarrow{a}$ on terms in $\mathbb{T}(BPA)$. The intended meaning of $x \xrightarrow{a} y$ is that process x may perform an a-action, and thereby evolve into process y.

In order to define the model, we have to extend the signature of BPA with an auxiliary constant δ. We denote the set of closed terms over this extended signature by $\mathbb{T}(BPA_\delta)$. In the ACP framework of Bergstra & Klop, δ is called **deadlock**. This name suggests a particular intuition about the behaviour of this process which is not in accordance with the interpretation of δ in our first model. Rather, δ plays the same role as NIL of CCS (see MILNER [M]) or STOP of CSP (see HOARE [H]).

The model we consider here, interprets · as **sequencing**: $x \cdot y$ starts with the execution of x, and if x can do no more actions, then execution of y starts. In all models that we present in this paper, the constant δ is characterised by being unable to perform any actions, i.e. $\delta \xrightarrow{a} x$ for no a,x. We write $x \not\rightarrow$ to denote that x has no outgoing transition (so we have $\delta \not\rightarrow$).

We define the predicates $\xrightarrow{a}$ inductively by means of the rules in table 2.

$$a \xrightarrow{a} \delta$$

$$\frac{x \xrightarrow{a} x'}{x+y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x+y \xrightarrow{a} y'}$$

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x \not\rightarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$$

TABLE 2. Action relations for BPA with sequencing.

2.5 A problem with this definition is the appearance of negative premisses. This makes that it is not immediately clear that there exists a distinguished transition relation agreeing with the rules. That such a relation exists in this case is due to the fact that the presence of an outgoing transition of a term only depends on the presence or absence of outgoing transitions from terms of a lower complexity. BLOOM, ISTRAIL & MEYER [BIM] (who also present the above rules for sequencing) observe that negative premisses are needed for the definition of this operator.

We turn the structure of action relations into a model for BPA by means of the notion of bisimulation (due to PARK [PA]).

2.6 DEFINITION. A binary relation R on process expressions is a (strong) **bisimulation** if it satisfies the so-called **transfer property**:

1. if $x \xrightarrow{a} y$ and $R(x,x')$ then there exists y' with $x' \xrightarrow{a} y'$ and $R(y,y')$ (for all labels a);
2. conversely, if $x' \xrightarrow{a} y'$ and $R(x,x')$ then there exists y with $x \xrightarrow{a} y$ and $R(y,y')$.

Two processes x,x' are called **bisimilar**, notation $x \leftrightarrows x'$, if there exists a bisimulation R with $R(x,x')$.

Then, the following theorems are standard:

2.7 THEOREM. Bisimulation is a congruence relation on $\mathbb{T}(BPA_\delta)$.

2.8 THEOREM. BPA is a complete axiomatisation of $\mathbb{T}(BPA)/\leftrightarrows$, i.e. for all terms s,t from $\mathbb{T}(BPA)$ we have

$$BPA \vdash s=t \quad \Leftrightarrow \quad \mathbb{T}(BPA)/\leftrightarrows \; \models s=t \quad \Leftrightarrow \quad s \leftrightarrows t.$$

2.9 Notice that theorem 2.8 only talks about terms from $\mathbb{T}(BPA)$, so terms not involving δ. As was already remarked by BERGSTRA & KLOP [BK1], axiom A4 is not valid any more on $\mathbb{T}(BPA_\delta)$ (using the valid axiom $\delta \cdot x = x$, we can derive $a \cdot b = (a + \delta) \cdot b = a \cdot b + \delta \cdot b = a \cdot b + b$). Therefore, if we want to extend theorem 2.8 to the case with δ, we have to restrict A4.

2.10 THEOREM. Let A4*, A6 and A7* be the following axioms:

$(ax + by + y')z = axz + (by + y')z$	A4*
$x + \delta = x$	A6
$\delta \cdot x = x$	A7*.

Then A1,2,3,4*,5,6,7* form a complete axiomatisation of $\mathbb{T}(BPA_\delta)/\leftrightarrows$, i.e. for all terms s,t from $\mathbb{T}(BPA_\delta)$ we have

$$A1,2,3,4^*,5,6,7^* \vdash s=t \quad \Leftrightarrow \quad \mathbb{T}(BPA_\delta)/\leftrightarrows \; \models s=t \quad \Leftrightarrow \quad s \leftrightarrows t.$$

If one takes a more denotational viewpoint, then Plotkin style rules are just a way to define function between labeled transition systems (process graphs). The last two rules of table 2, for instance, determine the operation of **sequencing**: given two process graphs g and h, $g \cdot h$ is the process graph obtained by appending a copy of h to each endnode of g. Sequencing is a simple and natural operation on process graphs. However, it turns out that in a setting with parallel composition and communication we often want to interpret the operator symbol $\cdot$ differently.

2.11 SEQUENTIAL COMPOSITION.
Consider a process $x \cdot y$, where x describes the behaviour of a system consisting of a number of processors which jointly perform some parallel computation. Then it may occur that at some point during the execution of x a state of **deadlock** is reached, i.e. all processors are waiting for each other, before the computation is finished. Usually, y is not allowed to start in such a situation, even though x has reached a state where no transitions are possible. Process y may start only when process x has terminated *successfully*. When we talk about sequential composition, we assume that there are two termination possibilities: successful termination and unsuccessful termination. The sequential composition of x and y starts with execution of x, followed by the execution of y upon successful termination of x. Now there are at different ways in which we can make this intuition more precise. We will present three alternatives, before focusing on the third alternative. Consecutively, we consider:

a. successful termination as a hidden signal (2.12);
b. successful termination as an attribute of actions (2.13);
c. successful termination with $\surd$-refinement (2.14).

2.12 SUCCESSFUL TERMINATION AS A HIDDEN SIGNAL.
Deadlock is considered as an unsuccessful form of termination. If deadlock is characterised by the absence of any possibility to proceed, it seems natural to introduce a special label to indicate successful termination. This special label is denoted $\surd$ (pronounced 'tick'). Thus, we will have an extra binary relation $\xrightarrow{\surd}$. Next, the behaviour of process $a \in A$ is described by the rules

$$a \xrightarrow{a} \varepsilon \qquad \varepsilon \xrightarrow{\surd} \delta.$$

Here, ε is a new constant symbol denoting the process that terminates immediately and successfully (ε first appears in KOYMANS & VRANCKEN [KV]). We see that the process a first performs an a-transition, and then terminates successfully. The process δ still has no outgoing transitions and therefore corresponds in this setting with the process which terminates immediately but unsuccessfully. Now what rules can we have for the sequential composition operator? First, we note that it would not be correct to have rules like

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x \xrightarrow{\surd} x'}{x \cdot y \xrightarrow{\surd} y}$$

because then we could derive things like

$$(a \cdot b) \cdot c \xrightarrow{a} (\varepsilon \cdot b) \cdot c \xrightarrow{\surd} c \xrightarrow{c} \varepsilon,$$

which are clearly in contrast with the intended semantics of the sequential composition operator. Hence $\surd$-events performed by the first argument of the $\cdot$ operator cannot remain visible.

One possible view on sequential composition, which is taken in CCS (see MILNER [M]), is that $\surd$-events do occur, but that they are 'hidden from our view'. This can be expressed formally by the following rules:

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \text{ (for } a \in A \cup \{\tau\}) \qquad \frac{x \xrightarrow{\surd} x'}{x \cdot y \xrightarrow{\tau} y}$$

Here τ is the silent move of MILNER [M]. Under this interpretation, the transitions of process $(a \cdot b) \cdot c$ are:

$$(a \cdot b) \cdot c \xrightarrow{a} (\varepsilon \cdot b) \cdot c \xrightarrow{\tau} b \cdot c \xrightarrow{b} \varepsilon \cdot c \xrightarrow{\tau} c \xrightarrow{c} \varepsilon \xrightarrow{\surd} \delta.$$

The introduction of τ leads to a number of difficult questions. For instance, should the process $(a \cdot b) \cdot c$ be considered equal to a process p with transitions

$$p \xrightarrow{a} q \xrightarrow{b} r \xrightarrow{c} s \xrightarrow{\surd} \delta.$$

In this paper we want to deal with *concrete* process algebra only, i.e. we do not want to consider the silent move and different alternatives for its axiomatisation and representation by means of action relations. Therefore, we will not pursue the above view on sequential composition any further in this paper.

2.13 SUCCESSFUL TERMINATION AS AN ATTRIBUTE OF ACTIONS.

In BRINKSMA [BR], a sequential composition operator is presented which is based on the idea that successful termination is a visible attribute of the last action of a process. Slightly simplified, this looks as follows: beside the actions in A, the set of labels also contains the elements of the set $A\surd = \{a\surd \mid a \in A\}$. The new action rules are $(a \in A)$:

$$a \xrightarrow{a\surd} \delta \qquad \frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x \xrightarrow{a\surd} x'}{x \cdot y \xrightarrow{a} y}.$$

This approach is comparable to the approach in VAN GLABBEEK [VG] (there, $a \xrightarrow{a\surd} \delta$ is written as $a \xrightarrow{a} \surd$). While this is a viable approach, the problem we have with it is, that there seems to be a mismatch with so-called 'True' concurrency and event structures. Many algebraic concurrency languages can be provided with a non-interleaved semantics. A reasonable criterion, put forward by DEGANO, DE NICOLA & MONTANARI [DDM] and OLDEROG [O], is that the interleaved semantics of a language must be *retrievable* from the non-interleaved semantics. Now consider the operator $\|$ of parallel composition without synchronisation. If we add such an operator to the current setting, the action rules will be

$$\frac{x \xrightarrow{a} x'}{x \| y \xrightarrow{a} x' \| y} \qquad \frac{x \xrightarrow{a\surd} x'}{x \| y \xrightarrow{a} y} \qquad \frac{y \xrightarrow{a} y'}{x \| y \xrightarrow{a} x \| y'} \qquad \frac{y \xrightarrow{a\surd} y'}{x \| y \xrightarrow{a} x}.$$

With these rules, the transition system for $a \| b$ becomes as shown in fig. 1.

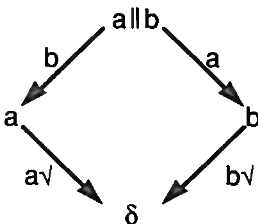


FIGURE 1.

It seems almost unavoidable that in a non-interleaved event structure semantics from which the above interleaving semantics is retrievable, there are 4 events $a, b, a\surd, b\surd$. Furthermore we do not see how to avoid that events a and b are conflicting, whereas event $b\surd$ is causally dependent on event a and event $a\surd$ is causally dependent on event b. But this would be in clear contradiction with the non-interleaved interpretation of $a \| b$ that one expects intuitively, where the two events a and b are not causally related.

Hence, we think that it will be difficult to give a non-interleaved event structure semantics which is compatible with this interpretation of sequential composition.

2.14 SUCCESSFUL TERMINATION WITH √-REFINEMENT.
A third view on sequential composition, the view we prefer, is that we have to do with an instance of **action refinement** as advocated e.g. by CASTELLANO, DE MICHELIS & POMELLO [CDP] and VAN GLABBEEK & GOLTZ [GG]. We again use √-labels to denote successful termination, and assume we have a process domain where √-events have no causal successors and are moreover not concurrent with any other event. This implies that a √-event, if it occurs, will always be the last action performed by a process. On such a domain the sequential composition of processes x and y can be implemented by *refining* every √-event of x to the process y. We refer to [GG] for a formal definition of refinement on the domain of event structures*. Here, we only present the action rules which correspond to the refinement view of sequential composition, in table 3.

$a \xrightarrow{a} \varepsilon$	$\varepsilon \xrightarrow{\surd} \delta$
$\dfrac{x \xrightarrow{u} x'}{x+y \xrightarrow{u} x'}$	$\dfrac{y \xrightarrow{u} y'}{x+y \xrightarrow{u} y'}$
$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	$\dfrac{x \xrightarrow{\surd} x' \quad y \xrightarrow{u} y'}{x \cdot y \xrightarrow{u} y'}$

TABLE 3. Sequential composition with action refinement.

In this table (and everywhere in the sequel), u stands for either a or √. This operational semantics can be found in BAETEN & VAN GLABBEEK [BG], only there $x \xrightarrow{\surd} \delta$ was written as $x\downarrow$. The present formulation is due to GROOTE & VAANDRAGER [GV].

Let $\mathbb{T}(BPA_{\delta\varepsilon})$ be the set of closed terms over the signature of BPA extended with the constants δ,ε. Since all rules in table 3 are in the *tyft* format of [GV], bisimulation is a congruence relation on $\mathbb{T}(BPA_{\delta\varepsilon})$. The rules of table 3 induce a model for BPA. In addition, we can also give an axiomatisation for the theory including the constants δ,ε. Let $BPA_{\delta\varepsilon}$ be the theory consisting of BPA together with the axioms in table 4.

$x + \delta = x$	A6
$\delta \cdot x = \delta$	A7
$\varepsilon \cdot x = x$	A8
$x \cdot \varepsilon = x$	A9

TABLE 4. Termination laws.

Thus, δ is the neutral element for alternative composition, ε is the neutral element for sequential composition. A7 is explained, since a deadlocked process can never perform any actions (notice the difference with A7*!). Note that if we added the dis-

* In fact, in [GG], actions are only refined by finite, conflict-free event structures. However, it can be easily seen that in case the actions which are refined have no causal successors, the definition of [GG] can be generalised to general event structures.

tributive law $x(y + z) = xy + xz$, then we could derive $ab = a(b + \delta) = ab + a\delta$, and so a process with no deadlock possibility would be equal to one that may deadlock, a clearly undesirable situation.

Now we have the following theorem, due to BAETEN & VAN GLABBEEK [BG].

2.15 THEOREM. $BPA_{\delta\varepsilon}$ is a complete axiomatisation of $\mathbb{T}(BPA_{\delta\varepsilon})/\underline{\leftrightarrow}$, i.e. for all terms s,t from $\mathbb{T}(BPA_{\delta\varepsilon})$ we have

$$BPA_{\delta\varepsilon} \vdash s=t \quad \Leftrightarrow \quad \mathbb{T}(BPA_{\delta\varepsilon})/\underline{\leftrightarrow} \models s=t \quad \Leftrightarrow \quad s \,\underline{\leftrightarrow}\, t.$$

In the next section, we will extend this last view on sequential composition to a setting with process creation.

3. PROCESS CREATION.

3.1 MOTIVATION.
In 2.14, we restricted our attention to a domain of processes where a $\surd$-event is always the last event in an execution. This was a natural restriction since $\cdot$ was interpreted as sequential composition and $\surd$ as successful termination. Now we would like to consider the operation of $\surd$-refinement on a more general domain where $\surd$-events still do not have causal successors but with the possibility that a $\surd$-event is concurrent with a non-$\surd$-event.

These more general processes can for instance arise if one has an operation **new**(x) which removes all $\surd$-events in a process and introduces a new $\surd$-event which is concurrent with the remaining events of x. In such a setting every process can perform at most one $\surd$-event in its lifetime but this is not necessarily the last event. If we interpret a as a process which first does an a-event followed by a $\surd$-event, and the operator symbol $\cdot$ as $\surd$-refinement, then we can stepwise construct the interpretation of a·(**new**(b·c)·d) as in fig. 2.

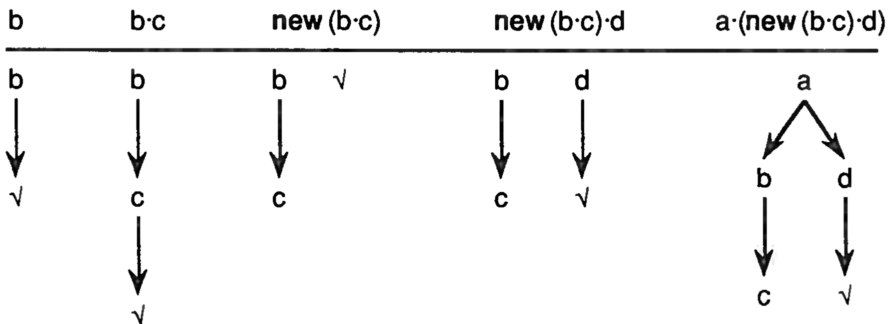


FIGURE 2.

One may think of fig. 2 as a graphical representation of a labeled prime event structure (see [W] or [GG] for the terminology). The arrows denote the causality relation. The traces of this process are

a b c d $\surd$	a b d c $\surd$	a b d $\surd$ c
a d $\surd$ b c	a d b $\surd$ c	a d b c $\surd$

The reader might notice that what we have achieved now is that we have informally given a semantics (essentially an event structure semantics) of a simple language

with process creation. Moreover, this semantics agrees with the intuitions concerning process creation that we presented in the introduction. In fact we claim that the semantics is compatible with the semantics given in AMERICA & DE BAKKER [AB] for a uniform and dynamic language (section 4). In the language of [AB], also alternative composition and μ-recursion are present. We have not described an interpretation of these operators on event structures because we do not want to become too technical here. Such an interpretation, however, is standard and described in many papers (see for instance [W]).

In [AB], the operator : is presented as an operator "which is able to decide dynamically whether it should act as sequential or parallel composition". We prefer a different intuition because we think that the operator : does not introduce a choice or conflict between sequential and parallel composition, but rather that it is a natural generalisation of the sequential composition operator · on a domain of processes where √ may occur in a non-final position.

A surprising thing about the above semantics is that, on the domain of event structures, we give exactly the *same* interpretation to the operator symbol · as in the case of sequential composition. The only difference is that the domain of processes is enlarged. When we work with the extended domain of processes, we will call this operation **continuation** and the √-event the **continuation action** (sequential composition and successful termination is not an appropriate terminology now).

3.2 CONTINUATION

We will now give Plotkin-style rules, which correspond to the above event structure semantics. It turns out that on this level we do have to change the rules for the · operator: since in a product $x{\cdot}y$, the process x may continue after y has started, we have to introduce an auxiliary operator $\Vert\!\!\!\ulcorner$ for describing those states where y has started but x is not yet finished. See table 5.

$\dfrac{x \xrightarrow{a} x'}{x{\cdot}y \xrightarrow{a} x'{\cdot}y}$	$\dfrac{x \xrightarrow{\surd} x' \quad y \xrightarrow{u} y'}{x{\cdot}y \xrightarrow{u} x' \Vert\!\!\!\ulcorner y'}$

TABLE 5. Continuation.

We can see, that the second rule here is a generalisation of the corresponding rule in table 3. There, if $x \xrightarrow{\surd} x'$, necessarily $x' \equiv \delta$, and the term $\delta \Vert\!\!\!\ulcorner x$ has the same transitions as x.

The reader may think there is a possibility missing here, viz.

$$\frac{x \xrightarrow{\surd} x' \xrightarrow{a} x''}{x{\cdot}y \xrightarrow{a} x'' \Vert\!\!\!\ulcorner y}.$$

However, this rule is not in accordance with our view on sequential composition with refinement of √-events: when y refines the √-event, any action in place of the √-event should involve an initial action of y. Moreover, the proposed rule leads to counter-intuitive behaviour: process x should behave the same as process $x{\cdot}\varepsilon$, but if x can perform √ and then a, then $x{\cdot}\varepsilon$ can also perform a before √ with the rule above.

3.3 PARALLEL COMPOSITION.

The operator $\Vert\!\!\!\ulcorner$ is just parallel composition with the additional restriction that only the process on the right-hand side may perform √-events. This operator is very similar to

the parallel composition operator in the theory ACP of [BK2,3]. In ACP, the parallel composition of x and y can perform a √-event only if both x and y can perform a √-event at the same time. When we compose processes x and y by means of our new combinator ⫦, the composition can do a √-event when y can do a √-event.

With respect to interleaving, ⫦ behaves as one would expect: if one component can perform a certain atomic action, the composition can also perform this action. In table 6, we present the action relation definition for ⫦.

$\dfrac{x \xrightarrow{a} x'}{x \Vdash y \xrightarrow{a} x' \Vdash y}$	$\dfrac{y \xrightarrow{u} y'}{x \Vdash y \xrightarrow{u} x \Vdash y'}$

TABLE 6. Parallel composition.

We should note that all our operators are defined on the domain of processes that is described above. Thus, any composition of processes that have at most one √-event in every execution path, again gives such a process. If one has no objection to operators that lead outside this domain, a symmetric parallel composition can be used, and x⫦y is represented by something like $\partial_{\{\surd\}}(x) \,\|\, y$ (where $\partial_{\{\surd\}}$ cancels all √'s).

In languages with process creation, parallel composition is mostly not included in the language. It is an auxiliary operator which is present only on a semantical level.

3.4 PROCESS CREATION.
The operator **new** can be defined by:

$$\mathbf{new}(x) = x \Vdash \varepsilon.$$

From this definition, it follows that **new** is characterised by the action rules in table 7.

$\mathbf{new}(x) \xrightarrow{\surd} x \cdot \delta$	$\dfrac{x \xrightarrow{a} x'}{\mathbf{new}(x) \xrightarrow{a} \mathbf{new}(x')}$

TABLE 7. Action rules for process creation.

We claim that this operator is essentially the same construct as the **new** of AMERICA & DE BAKKER [AB] (section 4).

3.5 EXAMPLE. The term a·**new**(b·c)·d determines the transition diagram in fig. 3.

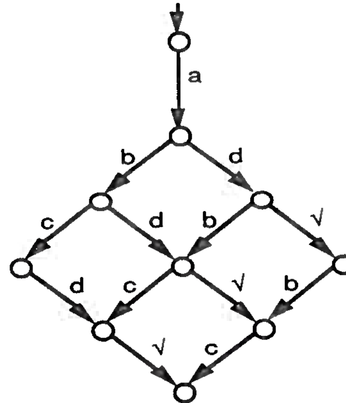


FIGURE 3.

3.6 COMMUNICATION.

Although the rules of tables 5-7 gave a simple and intuitive semantics to process creation, this semantics is not very practical. In any practical language with process creation there must be a possibility of *communication* between a newly created process and the rest of the system. Therefore, we add rules for $\Vert\!\!\!\ulcorner$ and $\cdot$ which express the possibility of communication.

Like in ACP, we have given a partial binary function γ on A, which is commutative and associative, the so-called **communication function**. If $\gamma(a,b) = c$, we say a and b communicate, and the result of the communication is c. If $\gamma(a,b)$ is undefined, we say that a and b do not communicate. In table 8, we present the new rules for $\Vert\!\!\!\ulcorner$ and $\cdot$.

We will not discuss here the consequences of the change in the action rules on the level of non-interleaved event structure semantics. It will be clear that $\cdot$ can no longer be interpreted as just refinement of tick-events. The construction will now introduce a large number of new events which describe possible synchronisations between the original and the new processes.

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{x \Vert\!\!\!\ulcorner y \xrightarrow{c} x' \Vert\!\!\!\ulcorner y'} \quad \text{if } \gamma(a,b) = c$$

$$\frac{x \xrightarrow{\surd} x' \xrightarrow{a} x'' \quad y \xrightarrow{b} y'}{x \cdot y \xrightarrow{c} x'' \Vert\!\!\!\ulcorner y'} \quad \text{if } \gamma(a,b) = c$$

TABLE 8. Communication

As before, a,b,c range over A, u ranges over $A \cup \{\surd\}$.

3.7 ENCAPSULATION.

As in ACP (see [BK2,3]), we have the **encapsulation operator** ∂_H (where H is a set of atomic actions), which blocks actions from H. This operator is used to block communications with the environment, and remove 'halves' of communication (actions that should communicate). The action relation definition is straightforward.

$$\frac{x \xrightarrow{u} x'}{\partial_H(x) \xrightarrow{u} \partial_H(x')} \quad \text{if } u \notin H$$

TABLE 9. Encapsulation.

3.8 BISIMULATION.
Since all the action rules presented so far are in the *tyft* format of GROOTE & VAANDRAGER [GV], we can conclude that bisimulation remains a congruence, also with respect to the new operators $\Vert\!\lceil$, $\cdot$, ∂_H. Thus, if $\mathbb{T}(PC)$ is the set of terms built with the signature of $BPA_{\delta\varepsilon}$ extended with these operators, then $\mathbb{T}(PC)/\leftrightarrows$ is a well-defined structure. In the next section, we will proceed to find a complete axiomatisation for this structure.

4. AXIOMATISATION.

4.1 AUXILIARY OPERATORS.
In order to give a finite axiomatisation for the structure $\mathbb{T}(PC)/\leftrightarrows$ defined in 3.8, we will need some auxiliary operators, comparable to the operators $\lfloor\!\lfloor$, $|$ in ACP (see [BK2,3]). Since our parallel composition operator is asymmetric, we will need not two but three auxiliary operators: $\Vert\!\lfloor$, $\rfloor\!\lceil$, $\lceil$. These three operators will form the three components of the merge operator $\Vert\!\lceil$: $\Vert\!\lfloor$, the **left-merge**, will give the possibilities that the left-hand side performs an event, $\rfloor\!\lceil$, the **right-merge**, gives the possibilities that the right-hand side performs an event (together, these two operators give the interleaving), and finally, $\lceil$, the **communication merge**, gives the possibilities that a communication action occurs between the two processes.

The axiomatisation to be presented also uses an additional auxiliary operator $\surd$. The process $\surd(x)$ starts with a $\surd$-event. Next the process x is performed from which however all $\surd$-events have been removed. When no confusion can occur, we will write $\surd x$ instead of $\surd(x)$.

4.2 SIGNATURE.
Now we will present the language for our **Algebra for Process Creation** (APC). As parameters of the language, we have a finite set A of atomic actions, and a partial binary function γ on A, which is commutative and associative. Then, we have constants a (for each $a \in A$), constants δ, ε, binary operators $+, \cdot, \Vert\!\lceil, \Vert\!\lfloor, \rfloor\!\lceil, \lceil$, a unary operator $\surd$, and unary operators ∂_H (for each $H \subseteq A$).

4.3 AXIOMS.
The axioms of APC are presented in table 10. There, $a,b \in A$, $H \subseteq A$, and x,y,z are arbitrary processes. Notice that the constant ε becomes definable, by axiom PC1.

$x + y = y + x$	A1	$x \Vert\!\lceil y = x \Vert\!\lfloor y + x \rfloor\!\rfloor\!\lceil y + x \lceil y$	PCM
$x + (y + z) = (x + y) + z$	A2		
$x + x = x$	A3	$\delta \Vert\!\lfloor x = \delta$	PCL1
$(x + y)z = xz + yz$	A4	$ax \Vert\!\lfloor y = a(x \Vert\!\lceil y)$	PCL2
$(xy)z = x(yz)$	A5	$\surd x \Vert\!\lfloor y = \delta$	PCL3
$x + \delta = x$	A6	$(x + y) \Vert\!\lfloor z = x \Vert\!\lfloor z + y \Vert\!\lfloor z$	PCL4
$\delta x = \delta$	A7		
$\varepsilon x = x$	A8	$x \rfloor\!\rfloor\!\lceil \delta = \delta$	PCR1
$x\varepsilon = x$	A9	$x \rfloor\!\rfloor\!\lceil ay = a(x \Vert\!\lceil y)$	PCR2
		$x \rfloor\!\rfloor\!\lceil \surd y = \surd(x \Vert\!\lceil y)$	PCR3
$\varepsilon = \surd\delta$	PC1	$x \rfloor\!\rfloor\!\lceil (y + z) = x \rfloor\!\rfloor\!\lceil y + x \rfloor\!\rfloor\!\lceil z$	PCR4
$\mathbf{new}(x)\cdot y = x \Vert\!\lceil y$	PC2		
$\surd x = \surd(x\delta)$	PC3	$ax \lceil by = \gamma(a,b)\cdot(x \Vert\!\lceil y)$	
$(\surd x)\cdot y = x \rfloor\!\rfloor\!\lceil y + x \lceil y$	PC4	if $\gamma(a,b)$ defined	PCC1
		$ax \lceil by = \delta$ if undefined	PCC2
$\partial_H(\delta) = \delta$	PCD1	$\surd x \lceil y = \delta$	PCC3
$\partial_H(ax) = a\cdot\partial_H(x)$ if $a \notin H$	PCD2	$x \lceil \surd y = \delta$	PCC4
$\partial_H(ax) = \delta$ if $a \in H$	PCD3	$(x + y) \lceil z = x \lceil z + y \lceil z$	PCC5
$\partial_H(\surd x) = \surd(\partial_H(x))$	PCD4	$x \lceil (y + z) = x \lceil y + x \lceil z$	PCC6
$\partial_H(x+y) = \partial_H(x) + \partial_H(y)$	PCD5		

TABLE 10. APC.

When we write a specification in APC, we only use a part of the signature, not the auxiliary operators. Formally, we can declare part of the signature to be *hidden*, as explained e.g. in BERGSTRA, HEERING & KLINT [BHK] or VAN GLABBEEK & VAANDRAGER [VGV].

The visible signature of APC is $\Sigma = \{a \mid a \in A\} \cup \{\delta, \varepsilon, +, \cdot, \mathbf{new}\} \cup \{\partial_H \mid H \subseteq A\}$, whereas the hidden signature contains $\Vert\!\lceil, \Vert\!\lfloor, \rfloor\!\rfloor\!\lceil, \lceil, \surd$. We will write all specifications in section 5 in the signature Σ.

4.4 LEMMA. We list some useful identities that can be derived from APC.

i.	$\delta \Vert\!\lceil x = x$	vii.	$\mathbf{new}(\varepsilon) = \varepsilon$
ii.	$\mathbf{new}(x) = x \Vert\!\lceil \varepsilon$	viii.	$(\surd x)\cdot\delta = \delta$
iii.	$\mathbf{new}(\delta) = \varepsilon$	ix.	$\surd(\surd x) = \varepsilon$
iv.	$\varepsilon \Vert\!\lfloor x = \delta$	x.	$(x \Vert\!\lceil y)\cdot z = x \Vert\!\lceil (y\cdot z)$
v.	$\varepsilon \lceil x = x \lceil \varepsilon = \delta$	xi.	$\surd x = x \rfloor\!\rfloor\!\lceil \varepsilon$.
vi.	$\delta \lceil x = x \lceil \delta = \delta$		

PROOF. Mostly straightforward. We give proofs for the difficult identities.

i. $\delta \Vert\!\lceil x = \delta \Vert\!\lfloor x + \delta \rfloor\!\rfloor\!\lceil x + \delta \lceil x = \delta + \delta \rfloor\!\rfloor\!\lceil x + \delta \lceil x =$
$= \delta \rfloor\!\rfloor\!\lceil x + \delta \lceil x = (\surd\delta)\cdot x = \varepsilon\cdot x = x$;

vi. $\delta \lceil x = \delta \lceil x + \varepsilon \lceil x = (\delta + \varepsilon) \lceil x = \varepsilon \lceil x = \delta$;

ix. $\surd(\surd x) = \surd((\surd x)\cdot\delta) = \surd\delta = \varepsilon$;

x. $(x \Vert\!\lceil y)\cdot z = (\mathbf{new}(x)\cdot y)\cdot z = \mathbf{new}(x)\cdot(y\cdot z) = x \Vert\!\lceil (y\cdot z)$;

xi. $\surd x = \surd x\cdot\varepsilon = x \rfloor\!\rfloor\!\lceil \varepsilon + x \lceil \varepsilon = x \rfloor\!\rfloor\!\lceil \varepsilon$.

Note that from axiom PC2 and lemma 4.4.ii it follows that the operators **new** and $\Vert\!\lceil$ can be defined in terms of each other. Also, by 4.4.xi, the auxiliary operator $\surd$ is definable in terms of $\rfloor\!\rfloor\!\lceil$ and ε.

4.5 ACTION RELATIONS.
We can also give action relation definitions for the auxiliary operators. We give the full set in table 11.

$$a \xrightarrow{a} \varepsilon \qquad \varepsilon \xrightarrow{\surd} \delta \qquad \surd x \xrightarrow{\surd} x\cdot\delta$$

$$\mathbf{new}(x) \xrightarrow{\surd} x\cdot\delta \qquad \frac{x \xrightarrow{a} x'}{\mathbf{new}(x) \xrightarrow{a} \mathbf{new}(x')}$$

$$\frac{x \xrightarrow{u} x'}{x+y \xrightarrow{u} x'} \qquad \frac{y \xrightarrow{u} y'}{x+y \xrightarrow{u} y'}$$

$$\frac{x \xrightarrow{a} x'}{x\cdot y \xrightarrow{a} x'\cdot y} \qquad \frac{x \xrightarrow{\surd} x' \quad y \xrightarrow{u} y'}{x\cdot y \xrightarrow{u} x' \Vert\!\lceil y}$$

$$\frac{x \xrightarrow{\surd} x' \xrightarrow{a} x'' \quad y \xrightarrow{b} y'}{x\cdot y \xrightarrow{c} x'' \Vert\!\lceil y'} \quad \text{if } \gamma(a,b) = c$$

$$\frac{x \xrightarrow{a} x'}{x \Vert\!\lceil y \xrightarrow{a} x' \Vert\!\lceil y \quad x \lfloor\!\lfloor y \xrightarrow{a} x' \Vert\!\lceil y}$$

$$\frac{y \xrightarrow{u} y'}{x \Vert\!\lceil y \xrightarrow{u} x \Vert\!\lceil y' \quad x \rfloor\!\rfloor\!\lceil y \xrightarrow{u} x \Vert\!\lceil y'}$$

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{x \Vert\!\lceil y \xrightarrow{c} x' \Vert\!\lceil y' \quad x \vert\!\lceil y \xrightarrow{c} x' \Vert\!\lceil y'} \quad \text{if } \gamma(a,b) = c$$

$$\frac{x \xrightarrow{u} x'}{\partial_H(x) \xrightarrow{u} \partial_H(x')} \quad u \notin H$$

TABLE 11. Action relations for APC.

Again, all these action rules are in *tyft* format, so bisimulation remains a congruence. Let us call the set of process expressions over this extended set of operators $\mathbb{T}$(APC). We will prove that the axiom system APC is a complete axiomatisation of $\mathbb{T}(\text{APC})/{\leftrightarrows}$. First, we will need some other results.

4.6 DEFINITION. We define some useful sets of terms.

i. The set of **bottom terms** is defined inductively by:
   - δ is a bottom term;
   - if t,s are bottom terms, then so are $a\cdot t$ and $t + s$.

ii. The set of **basic terms** is defined inductively by:
- δ is a basic term;
- if t is a bottom term, then $\surd t$ is a basic term;
- if t,s are basic terms, then so are at and $t + s$.

We see that a basic term is a closed term built from the signature δ, $+$, $\surd$, a, such that a $\surd$ occurs at most once in every execution sequence, and such that we have *only prefix multiplication* (as defined below).

4.7 DEFINITION. We say a term has **only prefix multiplication** if for each subterm of the form $t \cdot s$, t is an atomic action, and s is *not* an atomic action. This means that for these terms, instead of having constants a and general multiplication $\cdot$, we could also use a signature with only unary operators $a\cdot$. Notice that this is the usual situation in CCS (MILNER [M]) and CSP (HOARE [H]).

4.8 LEMMA. Let t be a basic term. Then there exists a bottom term t' such that APC $\vdash t \cdot \delta = t'$.

PROOF: Straightforward induction on the structure of basic terms.

4.9 THEOREM. (*Elimination Theorem*)
Let t be a closed APC-term. Then there exists a basic term t' such that APC $\vdash t = t'$.

PROOF: By an inductive argument, it is enough to prove the following claim:
Let q,q' be basic terms and let p be syntactically equal to δ, ε, a, $q+q'$, $q \cdot q'$, $q \| q'$, $q \lfloor\!\lfloor q'$, $q \rfloor\!\rfloor q'$, $q \mid q'$, $\surd q$, $new(q)$ or $\partial_H(q)$. Then there exists a basic term r such that APC $\vdash p=r$.
To prove this claim, we use induction on the *size* of a term. We define size inductively by:
- $size(\delta) = size(\varepsilon) = 1$
- $size(a) = 2$
- $size(t+t') = size(t \cdot t') = size(t \lfloor\!\lfloor t') = size(t \rfloor\!\rfloor t') = size(t \mid t') = size(t) + size(t')$
- $size(t \| t') = size(t) + size(t') + 1$
- $size(\surd t) = size(t) + 2$
- $size(new(t)) = size(t) + 5$
- $size(\partial_H(t)) = size(t) + 1$.

In the cases $p \equiv \delta$, $p \equiv q+q'$, we already have the required form.
- $p \equiv \varepsilon$: use PC1;
- $p \equiv a$: use A9 and PC1;
- $p \equiv q \cdot q'$: here we use induction on the structure of q. If $q \equiv \delta$, use A7; if $q \equiv a \cdot q''$, use A5 and the induction hypothesis; if $q \equiv \surd q''$, with q'' a bottom term, use PC4 and the induction hypothesis; if $q \equiv q'' + q^*$, use A4 and the induction hypothesis;
- $p \equiv q \| q'$: use PCM and the induction hypothesis;
- $p \equiv q \lfloor\!\lfloor q'$: here we use induction on the structure of q. If $q \equiv \delta$, use PCL1; if $q \equiv a \cdot q''$, use PCL2 and the induction hypothesis; if $q \equiv \surd q''$, use PCL3; if $q \equiv q'' + q^*$, use PCL4 and the induction hypothesis;
- $p \equiv q \rfloor\!\rfloor q'$: here we use induction on the structure of q'. If $q' \equiv \delta$, use PCR1; if $q' \equiv a \cdot q''$, use PCR2 and the induction hypothesis; if $q' \equiv \surd q''$, use PCR3 to write $p = \surd(q \| q'')$, by induction $p = \surd q^*$ for some basic q^*, by PC3 $p = \surd(q^* \cdot \delta)$, and then

by lemma 4.8 $p = \surd q^\circ$ for some bottom q°; if $q \equiv q'' + q^*$, use PCR4 and the induction hypothesis;

- $p \equiv q \upharpoonright q'$: by simultaneous induction on the structure of q and q'; left to the reader;
- $p \equiv \surd q$: use PC3 and lemma 4.8;
- $p \equiv \mathbf{new}(q)$: write $\mathbf{new}(q) = \mathbf{new}(q) \cdot \varepsilon = q \| \!\! \upharpoonright \varepsilon = q \| \!\! \upharpoonright \surd\delta$ and apply induction;
- $p \equiv \partial_H(q)$: similar to $p \equiv q \|\!\!\downharpoonright q'$; left to the reader.

4.10 THEOREM. (*Soundness Theorem*)
The structure $\mathbb{T}(APC)/\leftrightarrows$ is a model of APC.

PROOF: To prove the theorem, we need to check that each axiom of APC holds in $\mathbb{T}(APC)/\leftrightarrows$. As an example, consider axiom A5 (by far the most difficult one!).

Consider the relation R on $\mathbb{T}(APC)$, that relates all terms with themselves, and moreover relates each term of the form $(x \cdot y) \cdot z$ with $x \cdot (y \cdot z)$ (and vice versa), every term $(x \|\!\!\upharpoonright y) \cdot z$ with $x \|\!\!\upharpoonright (y \cdot z)$ (and v.v.), and every term $(x \|\!\!\upharpoonright y) \|\!\!\upharpoonright z$ with $x \|\!\!\upharpoonright (y \|\!\!\upharpoonright z)$ (and v.v.). We claim that R is a bisimulation on $\mathbb{T}(APC)$. To prove this, we need to check that the transfer property holds. This proof has a large number of cases. We will give some of these cases.

In principle, this part of the proof could have been done mechanically also. In fact, the tool ECRINS (see MADELAINE & DE SIMONE [MDS]) has been designed for doing this type of proofs. Unfortunately, ECRINS is not able to deal with Plotkin style rules with a lookahead of more than one, such as the third rule for the $\cdot$ operator.

Suppose from $(x \cdot y) \cdot z$, we can perform a step. This fact is proved by a proof following the rules for $\cdot$ in table 11. Now look at the last step in this proof.

CASE 1. The last step uses the first rule. Thus, $x \cdot y$ can do an a-step. Now look at the last step in the proof of this fact.

SUBCASE 1.1. This last step uses the first rule. Thus x can do an a-step, to a term x', say. We have $x \xrightarrow{a} x'$, and so the steps in the proof were $x \cdot y \xrightarrow{a} x' \cdot y$ and $(x \cdot y) \cdot z \xrightarrow{a} (x' \cdot y) \cdot z$. From the first rule and $x \xrightarrow{a} x'$, we derive immediately that $x \cdot (y \cdot z) \xrightarrow{a} x' \cdot (y \cdot z)$, and $(x' \cdot y) \cdot z$ and $x' \cdot (y \cdot z)$ are again related.

SUBCASE 1.2. This last step uses the second rule. Then, we must have $x \xrightarrow{\surd} x'$ and $y \xrightarrow{a} y'$, and so $x \cdot y \xrightarrow{a} x' \|\!\!\upharpoonright y'$ and $(x \cdot y) \cdot z \xrightarrow{a} (x' \|\!\!\upharpoonright y') \cdot z$. By the first rule, $y \xrightarrow{a} y'$ implies $y \cdot z \xrightarrow{a} y' \cdot z$, and by the second rule, using $x \xrightarrow{\surd} x'$, we derive $x \cdot (y \cdot z) \xrightarrow{a} x' \|\!\!\upharpoonright (y' \cdot z)$. Now $(x' \|\!\!\upharpoonright y') \cdot z$ and $x' \|\!\!\upharpoonright (y' \cdot z)$ are again related.

SUBCASE 1.3. This last step uses the third rule. Then, we must have $x \xrightarrow{\surd} x' \xrightarrow{a} x''$, $y \xrightarrow{b} y'$ and $\gamma(a,b) = c$, whence $x \cdot y \xrightarrow{c} x'' \|\!\!\upharpoonright y'$ and $(x \cdot y) \cdot z \xrightarrow{c} (x'' \|\!\!\upharpoonright y') \cdot z$. By the first rule, $y \xrightarrow{b} y'$ implies $y \cdot z \xrightarrow{b} y' \cdot z$, and by the third rule, using $x \xrightarrow{\surd} x' \xrightarrow{a} x''$, we derive $x \cdot (y \cdot z) \xrightarrow{c} x'' \|\!\!\upharpoonright (y' \cdot z)$. Now $(x'' \|\!\!\upharpoonright y') \cdot z$ and $x'' \|\!\!\upharpoonright (y' \cdot z)$ are again related.

CASE 2. The last step uses the second rule. Thus, $x \cdot y$ can do an $\surd$-step and $z \xrightarrow{u} z'$ for some u, z'. Now the only possibility that $x \cdot y$ can do an $\surd$-step, is as a result of rule 2, with $x \xrightarrow{\surd} x'$ and $y \xrightarrow{\surd} y'$, and so, we had $x \cdot y \xrightarrow{\surd} x' \|\!\!\upharpoonright y'$ and $(x \cdot y) \cdot z \xrightarrow{u} (x' \|\!\!\upharpoonright y') \|\!\!\upharpoonright z'$. By rule 2, using $y \xrightarrow{\surd} y'$ and $z \xrightarrow{u} z'$, we obtain $y \cdot z \xrightarrow{u} y' \|\!\!\upharpoonright z'$, and by rule 2 again, using $x \xrightarrow{\surd} x'$, we obtain $x \cdot (y \cdot z) \xrightarrow{u} x' \|\!\!\upharpoonright (y' \|\!\!\upharpoonright z')$. Now $(x' \|\!\!\upharpoonright y') \|\!\!\upharpoonright z'$ and $x' \|\!\!\upharpoonright (y' \|\!\!\upharpoonright z')$ are again related.

CASE 3. The last step uses the third rule. Thus, $x{\cdot}y$ can do a $\surd$-step followed by an a-step, $z \xrightarrow{b} z'$ and $\gamma(a,b) = c$ (for some a,b,c,z'). Now, as in case 2, $x{\cdot}y \xrightarrow{\surd}$ implies $x \xrightarrow{\surd} x'$ and $y \xrightarrow{\surd} y'$ and $x{\cdot}y \xrightarrow{\surd} x' \Vert\!\lceil y'$. This means that $x' \Vert\!\lceil y'$ can do an a-step. This must be the result of one of the three rules for $\Vert\!\lceil$.

SUBCASE 3.1. The first rule for $\Vert\!\lceil$ was used. Then, $x' \xrightarrow{a} x''$ for some x'', and so $x' \Vert\!\lceil y' \xrightarrow{a} x'' \Vert\!\lceil y'$ and $(x{\cdot}y){\cdot}z \xrightarrow{c} (x'' \Vert\!\lceil y') \Vert\!\lceil z'$. By the second rule for $\cdot$, using $y \xrightarrow{\surd} y'$ and $z \xrightarrow{b} z'$, we obtain $y{\cdot}z \xrightarrow{b} y' \Vert\!\lceil z'$. Then apply the third rule for $\cdot$, using $x \xrightarrow{\surd} x' \xrightarrow{a} x''$, to get $x{\cdot}(y{\cdot}z) \xrightarrow{c} x'' \Vert\!\lceil (y' \Vert\!\lceil z)$. Now $(x'' \Vert\!\lceil y') \Vert\!\lceil z$ and $x'' \Vert\!\lceil (y' \Vert\!\lceil z)$ are again related.

SUBCASE 3.2. The second rule for $\Vert\!\lceil$ was used. Then, $y' \xrightarrow{a} y''$ for some y'', and so $x' \Vert\!\lceil y' \xrightarrow{a} x' \Vert\!\lceil y''$ and $(x{\cdot}y){\cdot}z \xrightarrow{c} (x' \Vert\!\lceil y'') \Vert\!\lceil z'$. Apply the third rule for $\cdot$, using $y \xrightarrow{\surd} y' \xrightarrow{a} y''$ and $z \xrightarrow{b} z'$, to get $y{\cdot}z \xrightarrow{c} y'' \Vert\!\lceil z'$. Then use the second rule for $\cdot$ with $x \xrightarrow{\surd} x'$ to obtain $x{\cdot}(y{\cdot}z) \xrightarrow{c} x' \Vert\!\lceil (y'' \Vert\!\lceil z')$. Now $(x' \Vert\!\lceil y'') \Vert\!\lceil z'$ and $x' \Vert\!\lceil (y'' \Vert\!\lceil z')$ are again related.

SUBCASE 3.3. The third rule for $\Vert\!\lceil$ was used. Then a is the result of a communication, say between a' and a''. We find $x' \xrightarrow{a'} x''$, $y' \xrightarrow{a''} y''$, and so $x' \Vert\!\lceil y' \xrightarrow{a} x'' \Vert\!\lceil y''$ and $(x{\cdot}y){\cdot}z \xrightarrow{c} (x'' \Vert\!\lceil y'') \Vert\!\lceil z'$. Now use the third rule for $\Vert\!\lceil$ with $y \xrightarrow{\surd} y' \xrightarrow{a''} y''$ and $z \xrightarrow{b} z'$, to get $y{\cdot}z \xrightarrow{\gamma(a'',b)} y'' \Vert\!\lceil z'$. Now notice that by associativity of γ we have that $\gamma(a',\gamma(a'',b)) = c$. Applying this in the third rule for $\Vert\!\lceil$ again, with $x \xrightarrow{\surd} x' \xrightarrow{a'} x''$, leads to $x{\cdot}(y{\cdot}z) \xrightarrow{c} x'' \Vert\!\lceil (y'' \Vert\!\lceil z')$. Now $(x'' \Vert\!\lceil y'') \Vert\!\lceil z'$ and $x'' \Vert\!\lceil (y'' \Vert\!\lceil z')$ are again related.

Thus, we see that the transfer property holds from $(x{\cdot}y){\cdot}z$ to $x{\cdot}(y{\cdot}z)$. All information, needed to prove the converse implication is available above. In a similar fashion, we can prove the transfer property between $(x \Vert\!\lceil y){\cdot}z$ and $x \Vert\!\lceil (y{\cdot}z)$, and between $(x \Vert\!\lceil y) \Vert\!\lceil z$ and $x \Vert\!\lceil (y \Vert\!\lceil z)$. We conclude that the relation R is a bisimulation, and thus that law A5 holds in $\mathbb{T}(APC)/\underline{\leftrightarrow}$. Also, we have shown that the laws $(x \Vert\!\lceil y){\cdot}z = x \Vert\!\lceil (y{\cdot}z)$ and $(x \Vert\!\lceil y) \Vert\!\lceil z = x \Vert\!\lceil (y \Vert\!\lceil z)$ hold in $\mathbb{T}(APC)/\underline{\leftrightarrow}$.

Another interesting case in the soundness proof that we would like to mention is axiom PC4: $(\surd x){\cdot}y = x \rfloor\!\lceil y + x \lceil y$. In the soundness proof of this axiom (similar to, but much simpler than the proof for A5), we need the soundness of the law $x\delta \Vert\!\lceil y = x \Vert\!\lceil y$.

4.11 LEMMA. Let p be a basic term and let q be an APC-term.

i. If, for some $a \in A$ $p \xrightarrow{a} q$, then there exists a basic term q' with $size(q') < size(p)$ such that APC $\vdash p = a{\cdot}q' + p$ and APC $\vdash q = q'$;

ii. If $p \xrightarrow{\surd} q$, then there exists a bottom term q' with $size(q') < size(p)$ such that APC $\vdash p = \surd q' + p$ and APC $\vdash q = q'$.

PROOF: Straightforward induction on the structure of p.

4.12 THEOREM. *(Completeness Theorem)*
The axiom system APC is a complete axiomatisation of $\mathbb{T}(APC)/\underline{\leftrightarrow}$.

PROOF: Let $p,q \in \mathbb{T}$ with $p \underline{\leftrightarrow} q$. We have to prove that APC $\vdash p = q$. Since $\mathbb{T}(APC)/\underline{\leftrightarrow}$ is a model for APC, the elimination theorem 4.9 tells us that we only have to prove this for basic terms p,q. A simple argument gives that it is even enough to show that for basic terms p,q

$$p{+}q \underline{\leftrightarrow} q \;\Rightarrow\; \text{APC} \vdash p{+}q = q.$$

Assume $p{+}q \underline{\leftrightarrow} q$. We prove APC $\vdash p{+}q = q$ with induction on $size(p) + size(q)$.

- $p \equiv \delta$: use A1 and A7.
- $p \equiv a \cdot p'$: we have $p \xrightarrow{a} \varepsilon \cdot p'$. Since $p+q \leftrightarrow q$, there is a q' such that $q \xrightarrow{a} q'$ and $\varepsilon \cdot p' \leftrightarrow q'$. By lemma 4.11 there is a basic term q'' with $size(q'') < size(q)$, $q = a \cdot q'' + q$ and $q'' = q'$. Since $p' \leftrightarrow q''$, $p'+q'' \leftrightarrow q''$ and $q''+p' \leftrightarrow p'$. Thus, by induction, $p'+q'' = q''$ and $q''+p' = p'$, and hence $p' = q''$. But now we derive that $p+q = a \cdot p' + q = a \cdot q'' + q = q$.
- $p \equiv \surd p'$: this case is similar to the previous case.
- $p \equiv p'+p''$: since $p+q \leftrightarrow q$, we also have $p'+q \leftrightarrow q$ and $p''+q \leftrightarrow q$. By induction $p'+q = q$ and $p''+q = q$. Hence $p+q = p'+p''+q = p'+q+p''+q = q+q = q$.

4.13 STANDARD CONCURRENCY.
As a consequence of the completeness theorem, all equations that hold in the model $\mathbb{T}(APC)/\leftrightarrow$ can be proven to hold in APC for all closed terms. We list a few of these equations in table 12. A name often given to such sets of equations is *Standard Concurrency.*

$(x \Vdash y) \Vdash z = x \Vdash (y \Vdash z)$
$x \Vdash y \cdot \delta = y \Vdash x \cdot \delta$
$x \Vdash \delta = x \cdot \delta$

TABLE 12. Standard concurrency.

As consequences of these axioms, we mention the identities $(x \Vdash y) \Vdash z = (y \Vdash x) \Vdash z$ and $x \Vdash y = x \cdot \delta \Vdash y$.

Using these axioms, we can prove a variant of the well-known *Expansion Theorem,* that is very useful to break down the parallel composition of many processes. Since our parallel operator is not in the visible signature, we will not bother to state it here.

5. EXAMPLES.

In order to give some interesting examples of process definitions in APC, we will say a few words about recursive definitions (more can be found in [BK2,3, vG]).

5.1 DEFINITIONS. A **recursive specification** over APC is a (countable) set of equations $\{X = t_X \mid X \in V\}$, where V is a set of variables, and t_X is a term over APC, possibly using variables from V, but no other variables. There is exactly one equation $X = t_X$ for each variable X.

A **solution** of the recursive specification E in a certain domain is an interpretation of the variables of V as processes such that all equations of E are satisfied.

The **Recursive Definition Principle (RDP)** says that every recursive specification has a solution. In the action relation model of APC, RDP holds, if we add for each recursive specification $E = \{X = t_X \mid X \in V\}$ and for each $X \in V$ a constant $\langle X \mid E \rangle$ to the language, together with an action rule

$$\frac{\langle t_X \mid E \rangle \xrightarrow{u} y}{\langle X \mid E \rangle \xrightarrow{u} y}.$$

Here $\langle t_X \mid E \rangle$ denotes the term obtained from t_X by replacing each variable $Y \in V$ by $\langle Y \mid E \rangle$. These rules still fit the *tyft* format, and so bisimulation remains a congru-

ence. Moreover, one can see that all axioms of APC remain valid in the extended setting.

Recursive specifications are used to define (specify) processes. Note that not every recursive specification has a *unique* solution, for $\{X = X\}$ has every process as a solution. In order to get a class of processes with unique solutions, we formulate the condition of guardedness.

5.2 DEFINITIONS.

i. Let t be an APC-term, and X a variable in t. We call an occurrence of X in t **guarded** if X is preceded by an atomic action, i.e. t has a subterm of the form $a \cdot s$, with $a \in A$, and the X in question occurs in s. Otherwise, we call the occurrence of X **unguarded.**

ii. A recursive specification $\{X = t_X \mid X \in V\}$ is **guarded** if each occurrence of a variable in each t_X is guarded.

iii. The **Recursive Specification Principle (RSP)** is the assumption that every guarded recursive specification has at most one solution. We can prove that the extended model of APC satisfies RSP.

In the remainder of this section, we give a number of examples of recursive specifications in APC.

5.3 EXAMPLE 1: SYSTOLIC SORTING.

Systolic systems are characterised by a regular configuration of simple components or cells. Systolic systems have turned out to be useful in VLSI design (see KUNG [K]).

We describe a sorting machine, that can is always ready to input numbers (less than some upper bound N), and is always ready to output the smallest number it contains. This machine consists of a number of cells that each can contain two numbers, and will dynamically create more cells as they become needed. Our description is based on the description in KOSSEN & WEIJLAND [KW], where also a correctness proof can be found (in the setting of ACP_τ). Consider the configuration in fig. 4.

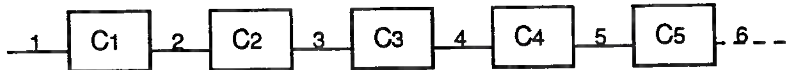


FIGURE 4.

The squares in fig. 4 represent the cells, the lines interconnecting them communication ports. We use the following atomic actions:

- $s_i(d)$ send number d along port i
- $r_i(d)$ read number d along port i
- $c_i(d)$ communicate number d along port i.

The communication function on these atomic actions is defined by: $\gamma(r_i(d), s_i(d)) = c_i(d)$, and γ is undefined on all other pairs.

Cell number i has three types of states, depending on whether it contains 0,1 or 2 numbers. The recursive specification of cell i is given in table 13.

Then, the sorting machine is given by:

$$SORT = \partial_H(C_1^0),$$

where $H = \{r_i(d), s_i(d) \mid i>1, d \le N\}$. Note that SORT has a guarded recursive specification.

$$C_i^0 = \sum_{d \le N} r_i(d) \cdot \mathbf{new}(C_{i+1}^0) \cdot C_i^1(d) + r_i(stop) + s_i(empty) \cdot C_i^0$$

$$C_i^1(d) = \sum_{e \le N} r_i(e) \cdot C_i^2(min(d,e), max(d,e)) + s_i(d) \cdot s_{i+1}(stop) \cdot C_i^0 \qquad (d \le N)$$

$$C_i^2(d,e) = \sum_{f \le N} r_i(f) \cdot s_{i+1}(e) \cdot C_i^2(min(d,f), max(d,f)) +$$

$$+ s_i(d) \cdot [\sum_{f \le N} r_{i+1}(f) \cdot C_i^2(min(e,f), max(e,f)) + r_{i+1}(empty) \cdot C_i^1(e)] \qquad (d \le e \le N)$$

TABLE 13. Systolic sorting.

5.4 EXAMPLE 2: QUEUE.
The specification of the unbounded (FIFO) queue is one of the recurring issues in process algebra. Examples of recursive specifications can be found in BAETEN & BERGSTRA [BB], VAN GLABBEEK & VAANDRAGER [VGV]. We will give two recursive specifications in APC involving the **new** construct: the first has an infinite number of equations, the second a finite number. To start with, we give the standard infinite specification of the queue in table 14. We denote the queue with contents σ (σ is a sequence of data elements, $\sigma \in D^*$ for some data set D) by Q_σ. λ is the empty sequence, d (for $d \in D$) also stands for a one element sequence, and $\sigma\rho$ denotes the concatenation of sequences σ and ρ.

$$Q_\lambda = \sum_{d \in D} in(d) \cdot Q_d$$

$$Q_{\sigma d} = \sum_{e \in D} in(e) \cdot Q_{e\sigma d} + out(d) \cdot Q_\sigma \qquad (\sigma \in D^*, d \in D)$$

TABLE 14. Queue, standard specification.

5.5 The second specification in APC will use an unlimited number of cells as in 5.3. This specification is inspired by a similar specification in DE SIMONE [DS]. Each cell can contain one data element; this element can be output when the permission for doing so is received: the permission go(i) will communicate with the *potential* output action pout(d,i) with as result the output out(d). Thus, we have a communication function γ given by:

$$\gamma(go(i), pout(d,i)) = out(d),$$

and γ is undefined otherwise. The definition of the cells and the queue is given in table 15. Note that this is a guarded specification. The encapsulation set is $H = \{go(i), pout(d,i) \mid d \in D, i \ge 1\}$ (D is the set of data elements).

$$C_i = \sum_{d \in D} in(d) \cdot \mathbf{new}(C_{i+1}) \cdot pout(d,i) \cdot go(i+1) \qquad (i \ge 1)$$

$$Q^1 = \partial_H(\mathbf{new}(C_1) \cdot go(1) \cdot \delta)$$

TABLE 15. Queue, first APC specification.

5.6 THEOREM. $Q^1 = Q_\lambda$.

PROOF: Define, for each $n \ge 1$ and each $\sigma \in D^*$, with $\sigma \equiv d_1 \ldots d_k$, the process R_σ^n by:

$$R_\sigma^n = \partial_H(C_{n+k} \Vert\!\!\lceil pout(d_1, n+k-1) \cdot go(n+k) \Vert\!\!\lceil \ldots \Vert\!\!\lceil pout(d_k, n) \cdot go(n+1) \Vert\!\!\lceil go(n) \cdot \delta).$$

(Note that we assume Standard Concurrency of 4.13 in this proof.) We will prove that, for each $n \geq 1$, $R^n_\sigma = Q_\sigma$. We do this by showing that the R^n_σ satisfy the specification in table 14. As a consequence, we derive

$$Q^1 = \partial_H(\mathbf{new}(C_1)\cdot go(1)\cdot\delta) = \partial_H(C_1 \Vert\!\!\lceil go(1)\cdot\delta) = R^1_\lambda = Q_\lambda.$$

Now the verification:

$$R^n_\lambda = \partial_H(C_n \Vert\!\!\lceil go(n)\cdot\delta) =$$

$$= \sum_{d\in D} in(d)\cdot\partial_H(\mathbf{new}(C_{n+1})\cdot pout(d,n)\cdot go(n+1) \Vert\!\!\lceil go(n)\cdot\delta) =$$

$$= \sum_{d\in D} in(d)\cdot\partial_H(C_{n+1} \Vert\!\!\lceil pout(d,n)\cdot go(n+1) \Vert\!\!\lceil go(n)\cdot\delta) =$$

$$= \sum_{d\in D} in(d)\cdot R^n_d.$$

Next, if $\sigma \equiv d_1 \ldots d_{k-1}$,

$$R^n_{\sigma d} = \partial_H(C_{n+k} \Vert\!\!\lceil pout(d_1,n+k-1)\cdot go(n+k) \Vert\!\!\lceil \ldots$$
$$\ldots \Vert\!\!\lceil pout(d_{k-1},n+1)\cdot go(n+2) \Vert\!\!\lceil pout(d,n)\cdot go(n+1) \Vert\!\!\lceil go(n)\cdot\delta) =$$

$$= \sum_{e\in D} in(e)\cdot\partial_H(\mathbf{new}(C_{n+k+1})\cdot pout(e,n+k)\cdot go(n+k+1) \Vert\!\!\lceil \ldots$$
$$\ldots \Vert\!\!\lceil pout(d,n)\cdot go(n+1) \Vert\!\!\lceil go(n)\cdot\delta) +$$
$$+ out(d)\cdot\partial_H(C_{n+k} \Vert\!\!\lceil pout(d_1,n+k-1)\cdot go(n+k) \Vert\!\!\lceil \ldots \Vert\!\!\lceil go(n+1) \Vert\!\!\lceil \delta) =$$

$$= \sum_{e\in D} in(e)\cdot\partial_H(C_{n+k+1} \Vert\!\!\lceil pout(e,n+k)\cdot go(n+k+1) \Vert\!\!\lceil \ldots$$
$$\ldots \Vert\!\!\lceil pout(d,n)\cdot go(n+1) \Vert\!\!\lceil go(n)\cdot\delta) +$$
$$+ out(d)\cdot\partial_H(C_{n+k} \Vert\!\!\lceil pout(d_1,n+k-1)\cdot go(n+k) \Vert\!\!\lceil \ldots \Vert\!\!\lceil go(n+1)\cdot\delta) =$$

$$= \sum_{e\in D} in(e)\cdot R^n_{e\sigma d} + out(d)\cdot R^{n+1}_\sigma.$$

Using RSP (see 5.2), we can show that the R^n_σ satisfy the specification in table 14.

5.7 THIRD SPECIFICATION OF QUEUE.

Next, we will give a finite recursive specification for the queue. In this specification, we will use action renaming. For each function $f: A \to A$, we introduce a unary operator ρ_f, that will rename atoms a into $f(a)$, and do nothing else. This operator is axiomatised in table 16. Action rules are quite easy to formulate.

$\rho_f(\delta) = \delta$
$\rho_f(ax) = f(a)\cdot\rho_f(x)$
$\rho_f(\surd x) = \surd(\rho_f(x))$
$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$

TABLE 16. Action renaming.

From BAETEN & BERGSTRA [BB] we know that the queue can be finitely specified in ACP plus renaming. In table 17, we give a finite specification in APC plus renaming.

$\mathrm{Cell} = \sum_{d \in D} \mathrm{in}(d) \cdot \partial_H(\mathbf{new}(\rho_f(\mathrm{Cell})) \cdot \mathrm{pre}(d) \cdot \mathrm{go})$
$Q^2 = \partial_H(\mathbf{new}(\rho_f(\mathrm{Cell}) \cdot \mathrm{go} \cdot \delta)$

TABLE 17. Queue, second specification.

Here, we use the renaming function f that renames each pre(d) into near(d), and leaves all other atoms unchanged. The communication function is specified by $\gamma(\mathrm{go}, \mathrm{near}(d)) = \mathrm{out}(d)$ (undefined otherwise). The encapsulation set is $H = \{\mathrm{go}\} \cup \{\mathrm{near}(d) \mid d \in D\}$.

In order to see that the specification in table 17 indeed describes a FIFO-queue, it might be illustrative to consider fig. 5.

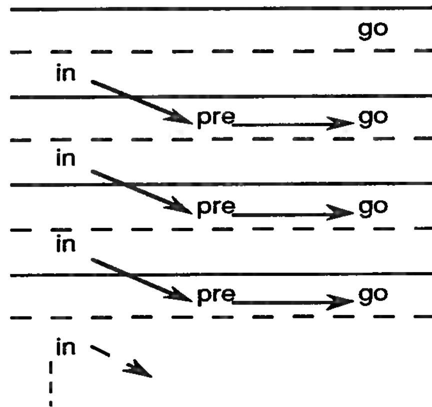


FIGURE 5.

In this diagram, we have abstracted from data d in actions in(d), out(d), etc. With arrows the 'causal' links between events are denoted. A black line stands for an encapsulation operator ∂_H and a dashed line for a renaming operator ρ_f.

One may imagine that in an execution, events 'bubble' upwards until they have passed through the surface of the topmost encapsulation line. An events cannot move before all its causal predecessors have occurred. A pre-event can pass through both types of lines. However, when it passes through a dashed line, it is renamed into a near-event. near-events and go-events are blocked by a black line. The synchronisation of a near-event and a go-event, however, gives a out-event. out-events, like in-events, can pass through both types of lines.

Along these same lines, we can give a recursive specification for the stack in APC.

5.8 THEOREM. $Q^2 = Q_\lambda$.

PROOF (sketch): Similar to 5.6. We define processes S_σ, that satisfy the specification in table 14. The S_σ are defined by using auxiliary processes T_σ, that, in turn, are defined inductively:

$$T_\lambda = \mathrm{Cell}$$
$$T_{d\sigma} = \partial_H(\rho_f(T_\sigma \Vert\!\!\lfloor\, \mathrm{pre}(d) \cdot \mathrm{go}))$$
$$S_\sigma = \partial_H(\rho_f(T_\sigma \Vert\!\!\lfloor\, \mathrm{go} \cdot \delta)).$$

The proof that the S_σ satisfy the specification in table 14 makes use of *alphabet information*. For more information on this type of argument, see BAETEN, BERGSTRA & KLOP [BBK].

5.9 EXAMPLE 3: BAG.
Along the same lines as for queue, we can give a simple recursive specification for the bag (an *unordered* channel; a state of the bag can be considered as a multiset of objects). We give the recursive specification in table 18, without further comment.

$\mathrm{Bag} = \sum_{d \in D} \mathrm{in}(d) \cdot \mathrm{new}(\mathrm{out}(d)) \cdot \mathrm{Bag}$

TABLE 18. Bag.

ACKNOWLEDGEMENT. The idea for an event structure semantics for the new operator arose following an inspiring discussion with Henk Goeman.

REFERENCES.

[A] P. AMERICA, *Definition of the programming language Pool-T*, ESPRIT project 415, nr. 0091, Philips Research Laboratories, Eindhoven 1985.
[AB] P. AMERICA & J.W. DE BAKKER, *Designing equivalent semantic models for process creation*, TCS 60, 1988, pp. 109-176.
[BB] J.C.M. BAETEN & J.A. BERGSTRA, *Global renaming operators in concrete process algebra*, I&C 78, 1988, pp. 205-245.
[BBK] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP, *Conditional axioms and α/β-calculus in process algebra*, in: Proc. IFIP Conf. Formal Descr. of Progr. Concepts III (M. Wirsing, ed.), North-Holland, Amsterdam 1987, pp. 53-75.
[BG] J.C.M. BAETEN & R.J. VAN GLABBEEK, *Merge and termination in process algebra*, in: Proc. 7th FST&TCS, Pune (K.V. Nori, ed.), Springer LNCS 287, 1987, pp. 153-172.
[B] J.A. BERGSTRA, *A process creation mechanism in process algebra*, in: Applications of Process Algebra (J.C.M. Baeten, ed.), CWI Monograph 8, North-Holland, Amsterdam 1989, pp. 81-88.
[BHK] J.A. BERGSTRA, J. HEERING & P. KLINT, *Module algebra*, report CS-R8844, Centre for Math. & Comp. Sci., Amsterdam 1988. To appear in JACM.
[BK1] J.A. BERGSTRA & J.W. KLOP, *Fixed point semantics in process algebras*, report IW 206, Math. Centre, Amsterdam 1982.
[BK2] J.A. BERGSTRA & J.W. KLOP, *Process algebra for synchronous communication*, I&C 60, 1984, pp. 109-137.
[BK3] J.A. BERGSTRA & J.W. KLOP, *Process algebra: specification and verification in bisimulation semantics*, in: Math. & Comp. Sci. II (eds. M. Hazewinkel, J.K. Lenstra & L.G.L.T. Meertens), CWI Monograph 4, North-Holland, Amsterdam 1986, pp. 61-94.
[BIM] B. BLOOM, S. ISTRAIL & A.R. MEYER, *Bisimulation can't be traced: preliminary report*, in: Proc. 15th POPL, San Diego Ca. 1988, pp. 229-239.
[BC] G. BOUDOL & I. CASTELLANI, *Permutation of transitions: an event structure semantics for CCS and SCCS*, report 798, INRIA, Sophia Antipolis 1988. To appear in Proc. REX School (J.W. de Bakker, W.-P. de Roever & G. Rozenberg, eds.), Springer LNCS.
[BR] E. BRINKSMA, *On the design of extended LOTOS*, Ph.D. thesis, University of Twente, 1988.

[CDP] L. CASTELLANO, G. DE MICHELIS & L. POMELLO, *Concurrency vs. interleaving: an instructive example*, Bulletin of the EATCS 31, 1987, pp. 12-15.

[DDM] P. DEGANO, R. DE NICOLA & U. MONTANARI, *A distributed operational semantics for CCS based on condition/event systems*, Acta Informatica 26 (1/2), 1988, pp. 59-91.

[vG] R.J. VAN GLABBEEK, *Bounded nondeterminism and the approximation induction principle in process algebra*, in: Proc. STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), Springer LNCS 247, 1987, pp. 336-347.

[GG] R.J. VAN GLABBEEK & U. GOLTZ, *Equivalence notions for concurrent systems and refinement of actions*, report 366, GMD, Sankt Augustin 1989.

[vGV] R.J. VAN GLABBEEK & F.W. VAANDRAGER, *Modular specifications in process algebra (with curious queues)*, report CS-R8821, Centre for Math. & Comp. Sci., Amsterdam 1988.

[GV] J.F. GROOTE & F.W. VAANDRAGER, *Structured operational semantics and bisimulation as a congruence*, report CS-R8845, Centre for Math. & Comp. Sci., Amsterdam 1988.

[H] C.A.R. HOARE, *Communicating sequential processes*, Prentice-Hall 1985.

[KW] L. KOSSEN & W.P. WEIJLAND, *Correctness proofs for systolic algorithms: palindromes and sorting*, in: Applications of Process Algebra (J.C.M. Baeten, ed.), CWI Monograph 8, North-Holland, Amsterdam 1989, pp. 89-125.

[KV] C.J.P. KOYMANS & J.L.M. VRANCKEN, *Extending process algebra with the empty process ε*, report LGPS 1, Faculty of Philosophy, State University of Utrecht 1985.

[K] K.T. KUNG, *Let's design algorithms for VLSI systems*, in: Proc. Conf. on VLSI: architecture, design, fabrication, California Institute of Technology, 1979.

[MDS] E. MADELAINE & R. DE SIMONE, *ECRINS un laboratoire de preuve pour les calculs de processus*, report R.R. 672, INRIA, Sophia Antipolis 1987.

[M] R. MILNER, *A calculus for communicating systems*, Springer LNCS 92, 1980.

[O] E.-R. OLDEROG, *Operational Petri net semantics for CCSP*, in: Advances in Petri Nets (G. Rozenberg, ed.), Springer LNCS 266, 1987, pp. 196-223.

[PA] D.M.R. PARK, *Concurrency and automata on infinite sequences*, in: Proc. 5th GI (P. Deussen, ed.), Springer LNCS 104, 1981, pp. 167-183.

[PL] G.D. PLOTKIN, *A structural approach to operational semantics*, report DAIMI FN-19, Comp. Sci. Dept., Aarhus University 1981.

[DS] R. DE SIMONE, *Higher-level synchronising devices in MEIJE-SCCS*, TCS 37, 1985, pp. 245-267.

[SS] S.A. SMOLKA & R.E. STROM, *A CCS semantics for NIL*, in: Formal Descr. of Progr. Concepts III (M. Wirsing, ed.), North-Holland, Amsterdam 1987, pp. 347-367.

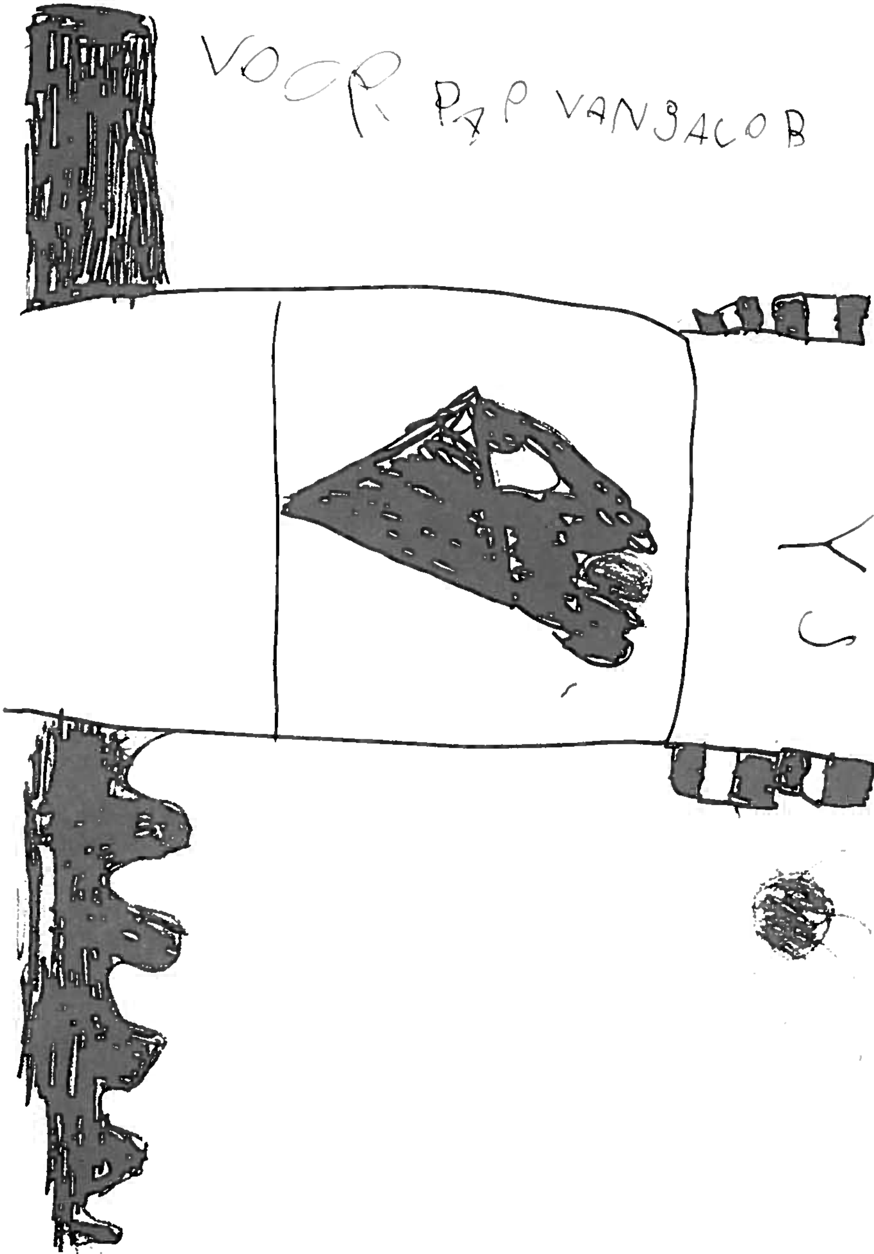
[VA] F.W. VAANDRAGER, *Process algebra semantics of POOL*, in: Applications of process algebra (J.C.M. Baeten, ed.), CWI Monograph 8, North-Holland, Amsterdam 1989, pp. 173-236.

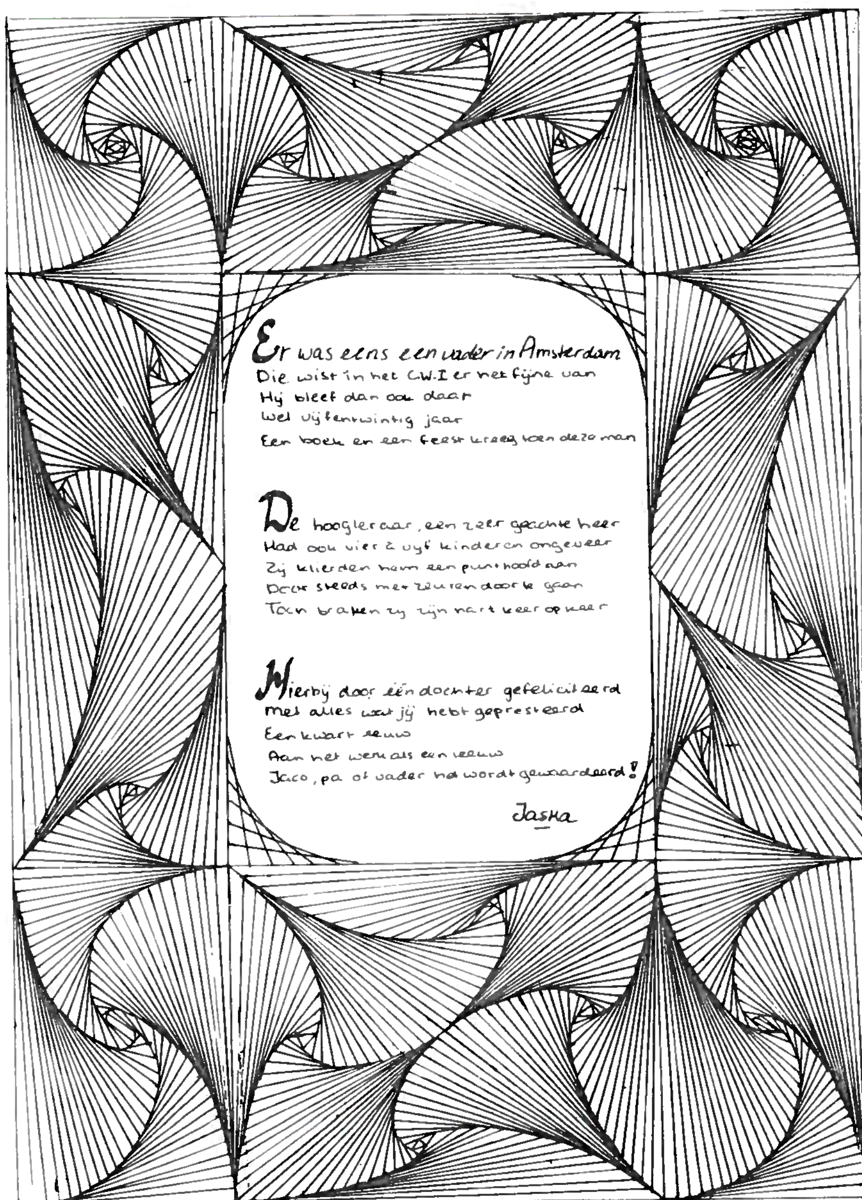
[W] G. WINSKEL, *Event structures*, in: Petri Nets: Applications and Relationships to Other Models of Concurrency, Bad Honnef 1986 (W. Brauer, W. Reisig & G. Rozenberg, eds.), Springer LNCS 255, 1987, pp. 325-392.

25 JAAR

VOOR PAP VAN

JACOB


VOOR PAP VAN JACOB


Er was eens een vader in Amsterdam
Die wist in het C.W.I er het fijne van
Hij bleef dan ook daar
Wel vijfentwintig jaar
Een boek en een feest kreeg toen deze man
De hoogleraar, een zeer geachte heer
Had ook vier à vijf kinderen ongeveer
Zij klierden hem een punthoofd aan
Door steeds met zeuren door te gaan
Toch braken zij zijn hart keer op keer
Hierbij door één dochter gefeliciteerd
Met alles wat jij hebt gepresteerd
Een kwart eeuw
Aan het werk als een leeuw
Jaco, pa of vader het wordt gewaardeerd!
Jaska

VOOR PAP VAN Lisa

BMACP

J.A. Bergstra
Programming Research Group, University of Amsterdam
Department of Philosophy, University of Utrecht

J.W. Klop
Department of Software Technology, Centre for Mathematics and Computer Science
Department of Computer Science, Free University Amsterdam

ABSTRACT. A new axiom system BMACP combines the axiom systems BMA (basic module algebra) and ACP (algebra of communicating processes) in such a way that the addition operator (+) which stands for module combination in BMA and for alternative composition of processes in ACP are both realised by the same operator (+) in BMACP. A closed process-module expression in BMACP denotes a process together with a specification of some auxiliary processes and all of these may be parametrised by some other processes.

This paper was written in honour of J.W. de Bakker on the occasion of the 25th anniversary of his association with the CWI. This work has been sponsored in part by ESPRIT project 432 METEOR.

1. MIXING PROCESS EXPRESSIONS AND DECLARATIONS

A topic that currently receives some interest is the design of specification languages in which process algebra in some form is used to describe dynamic aspects of target systems. Several difficulties have to be mastered mainly due to the fact that these specification languages combine declarative parts (declarations of data types, action alphabets, process names, state operators) and imperative parts (process expressions):

(i) How to syntactically combine process descriptions and data type descriptions? (LOTOS [Br 88] provides a sense of direction, PSF_d [M&V 88] redesigns a part of LOTOS in terms of ACP [BK 84] and ASF [BHK 89], further CRL [SP 89] a language that is being defined in the RACE project SPECS is yet another combination.)

(ii) How to define the meaning of modular algebraic specifications involving process descriptions (noticing the fact that for infinite processes bisimulation semantics is not obtained as an initial algebra semantics of their finite equational process specifications in process algebra, this in contrast with the case for finite processes)?

(iii) How to design formalisms combining process and data descriptions in such a way that the term rewriting paradigm is exploited best for both data and process specifications?

(iv) To what extent is equational logic sufficiently expressive for specification languages that combine data and process definitions?

(v) What are appropriate scope rules for local declarations of actions, process names and data?

(vi) What, if any, is the relationship between encapsulation and abstraction in process algebra and data abstraction (information hiding) as it occurs in static data type specifications.

2. MODULE ALGEBRA, A WAY OUT FOR THE LAZY THEORIST

The main dogma of module algebra is that the modularization constructions for specification languages fail to have a standard semantics and even fail to possess a simple model that reflects the intuition of the naive practitioner in the way the trace model describes the behavior of sequential and concurrent systems. Thus in [BHK 86] various models for module algebra have been found which are all quite different. In fact it is rather the absence of a single convincing model than the presence of various nice models which is the message of [BHK 86].

Modular specifications of processes can be described by embedding them in a module algebra on top of the process algebra. For this to be useful there is no need whatsoever for a 'nice' compositional semantics of the module algebra that assigns to modular process specifications a meaning that is intrinsic in terms of processes, behaviors, transition systems or something similar. Module algebra suggests the following minimalistic approach to the semantic problems connected with modularized specification languages.

(i) Introduce an algebraic syntax that allows to describe the module expressions that may be relevant for a particular (fragment of) a specification language that is being studied. Denote the resulting module expressions with ME. Define which expressions are normal forms, thus defining a class NMF(ME). The normal forms should show little or none of the modularisation mechanisms.

(ii) Provide a calculus C (or rather an equational proof system) that allows to transform each module expression to a normal form in the sense of (i).

(iii) Define a meaning for the module expressions in normal form in terms of the underlying concepts. It may be the case that only a part NMF* of NMF(ME) can be provided with a proper meaning. NMF* contains the 'nice' normal forms. The meaning of B in NMF* is denoted with SEM(B). It may well be rather difficult to pin down exactly the nice cases NMF*.

(iv) Prove that this meaning for NMF* is independent from the way in which the normal form was obtained by equational reasoning in the calculus C. Thus if A in ME has two normal forms B1 and B2 then B1 is part of NMF* if and only B2 is and in that case SEM(B1) = SEM(B2). This fact will then be referred to as the soundness of the calculus C. (One may consider the fact that C admits transformation to normal forms as completeness.)

The virtue of this procedure is that the meaning of a complex module expression A can be understood as follows:

(i) If A is an entire system then find a normal form B for A. If B is in NMF* then SEM(B) is the semantics of A.

(ii) If A is a component that has to be put in a context C[.] then one may safely simplify A using the rules of the calculus without the risk of changing SEM(C[A]) if it is defined. In particular A can be reduced to a normal form B and its intuitive meaning is then λ C[.]. C[B].

For the semantic description of complex specification languages the above program is much more easily carried out than the definition of an abstract and compositional semantics that provides a nice and natural mathematical meaning to every expression in ME. Of course this simply means that one (temporarily) withdraws from mathematical semantics and focusses on term models.

2.1. Fixed points and the μ-construct

We introduce an extension BMACP of ACP from [BK 84] by adding to it a part of the module algebra BMA of [BHK 86]. This extension should help in the solution of the problems (i), (ii) and (v) mentioned in 1 above. Further the axiom system BMACP sheds light on the status of the μ-operator in process algebra. We expand on the latter point first. The authors started working in process algebra after a talk by De Bakker held in Utrecht in June 1982. Then he posed the problem about the existence of fixed points of unguarded recursion equations in the topological model of processes that he developed in cooperation with Zucker (see [deBZ 82]). Our approach as reported in [BK 82] was to restrict attention to the case with a finite set of atomic actions and to define the equational axiom system PA involving alternative composition, sequential composition, merge and an auxiliary function named left merge. This axiom system PA has an initial algebra A_ω. This model allows finite projections A_n in which all processes are cut off after n steps. These finite projections are also models of PA. Then it was shown that the family of algebras A_n has a projective limit denoted with A_∞ which is a model of PA and more importantly can be isomorphically embedded in the aforementioned model of De Bakker and Zucker. By proving that all recursion equations can be solved in all finite projection algebras A_n it follows immediately that all equations are solvable in A_∞ and the problem posed by De Bakker was solved.

Now one easily proves that an unguarded equation of the form $X = P(X)$ has more then one solution in A_∞. In our view the algebra A_∞ stands out as a typical example of a process algebra. We have not yet found a natural way, however, to single out one canonical solution of (unguarded) recursion equations in A_∞. It follows that the definition of a μ-construct in the style of CCS [M 80] poses difficulties in A_∞. Indeed $\mu X . P(X)$ must denote some 'fixed' solution of the equation $X = P(X)$. Exactly because of these problems the first author strongly feels that the μ-construct should not be a part of process algebra. (Due to his history in lambda calculus the second author has a genuine sympathy for the binding mechanism of the μ-construct, however.) The axiom

system BMACP that we will describe allows the use of fixed point definitions within process expressions without any presuppositions concerning the (canonical) solvability of equations in a model.

It should be added immediately that the semantic pretentions of BMACP are minimal. Thus there is no implication that we can assign a meaning to fixed point constructions more often or in any better way than is possible with other theories. Rather conversely the meaning of the fixed point constructions is left completely open.

It should be noticed that the process theory of De Bakker and Zucker combines aspects of earlier process theories such as CSP [H 78] and CCS [M 80] by simultaneously using general sequential composition (taken from CSP) and a branching time semantics (taken from CCS). The main difference between CCS and the axiom system ACP from [BK 84] is due to the fact that in ACP general sequential composition is a primitive operator and not just prefix multiplication. In this sense ACP is an indeed an algebraic form of the model proposed in [deBZ 82].

This paper is not the first one that combines process algebra and module algebra. In [vGV 88] a much more natural combination is used. There the export mechanism of module algebra is used to hide the auxiliary functions left merge and communication merge after they have been helpful for the specification of the merge. This is needed if a transition to more abstract semantic models is to be made by adding equations that may be inconsistent in the presence of these auxiliary operators but are consistent in their absence.

2.2. Processes involving declarations

We can imagine that readers find our proposed BMACP rather grotesque and perhaps even clumsy. One of our purposes with the development of ACP is to provide extensions of ACP which are helpful for the design of specification languages. It turns out that each particular specification language design project leads to a multitude of matters of marginal scientific interest about which so-called 'design decisions' have to be made. The extension of ACP to BMACP addresses one of these marginal and somewhat unattractive aspects: scope rules for local process declarations in modular process specifications. The problem involved in these scope rules is easily illustrated with the main equation about δ in ACP:

$$\delta \cdot X = \delta$$

If X contains a process declaration with an unbounded scope then the equation is simply false. In BMACP we replace this equation by

$$\delta \cdot X = \partial(X).$$

Here A is the collection of all atomic actions and ∂ (process removal) replaces all atoms in X on an active position by δ. Now ∂ leaves the declarations of X unaffected, in fact it computes the declarations of X. ACP can be retrieved from BMACP by assuming the absence of declarations in

process expressions (i.e. $\partial(X) = \delta$). (Notice that the axiom system PA is obtained in a similar way from ACP. If all communications lead to δ then ACP reduces to PA_δ. So this way of extending a process algebra axiomatisation has precedents, be it that BMACP seems to have no models that can be viewed as a process algebra in any useful sense.)

2.3. A module algebra of recursive process specifications

Similar to the module algebra of [BKH 86] we may view individual recursion equations as atomic modules and import them to a class of modules (or rather module expressions) by an embedding operator $\langle\rangle$. Modules are then composed by means of an infix operator + which is then overloaded to denote the combination of specification modules as well as alternative composition of processes. Thus a list of defining equations will be represented as a combination of its elements (put between angular brackets). An example is as follows:

$$\langle P = a \cdot Q\rangle + \langle Q = b \cdot Q\rangle.$$

Now in many cases one is interested in process expressions that are written with use of the operators of process algebra and names of recursively defined processes. We propose that such process expressions may occur in sums besides recursion equations between angular brackets. From a different viewpoint one may say that we extend the algebra of process expressions with expressions of the form $\langle P = X\rangle$ with P a process name and X a process expression. A typical example is:

$$a \cdot Q + \langle Q = b \cdot Q\rangle.$$

If two such process expressions are combined with + their process parts are added in the way of process algebra and their declaration parts are added in the way of module algebra. If two such processes are merged then their process parts are merged and again the declaration parts are combined in the way of process algebra. The process part of a pure declaration like $\langle R = b \cdot R\rangle$ is just δ.

It will turn out that BMA contains an erasing axiom $X = X + (V \,\square\, X)$ which will be replaced by $X = X + \partial(V \,\square\, X)$. Notice that for purely declarative module expressions the identity $\partial(X) = X$ holds.

3. BMACP(A,γ,N), THE SIGNATURE

Let A be a finite collection of atomic actions not including δ. A_δ denotes the union of A and $\{\delta\}$. N is an infinite alphabet of process names disjoint from A. The function γ denotes the communication function which is a partial mapping of type $A \times A \rightarrow A$ which is associative and commutative. We will first describe the signature of the axiom system BMACP(A,γ,N). The role of the parameters A and N is to provide collections of constants for the sorts AP(atomic processes)

and PN (process names). The most important sort (type of interest) is the sort PM of process modules which at the same time plays the role of the processes of ACP and of the modules in BMA. The sort SIG wil contain signatures of process modules. In our setting such a signature is nothing more than the finite set of process names that are declared in the module.

```
begin signature Σ_BMACP(A, N)
    sorts     PN                      (process names)
              AA                      (atomic actions)
              AP                      (atomic processes)
              PM                      (process modules)
              ER                      (elementary renamings = pairs of process names)
              SIG                     (signatures = finite subsets of PN)
    constants
              a → AA                  (for a ∈ A, atomic action)
              δ → AP                  (deadlock)
              Ø → SIG                 (empty signature)
              n → PN                  (for n ∈ N, process name)
              Id → ER                 (identity)
    functions (canonical embeddings)
              i_aa_ap: AA → AP        (embedding of actions into atomic processes)
              i_ap_pm: AP → PM        (embedding of atomic processes in PM)
              i_pn_sig: PN → SIG      (embedding of process names in SIG)
              i_pn_pm: PN → PM        (embedding of process names in PM)
    functions (inherited from process algebra)
              +: PM x PM → PM         (alternative composition, serving as
                                      module combination as well)
              ·: PM x PM → PM         (sequential composition)
              ||: PM x PM → PM        (merge)
              ⌊⌊: PM x PM → PM        (left merge)
              |: PM x PM → PM         (communication merge)
              ∂_H: PM → PM            (encapsulation, for each subset H of A)
    functions (inherited from module algebra)
              +: SIG x SIG → SIG      (combination of signatures)
              ∩: SIG x SIG → SIG      (intersection of signatures)
              Σ: PM → SIG             (visible signature)
              T: SIG → PM             (embedding of signatures in PM)
              □: SIG x PM → PM        (export)
              r: PN x PN → ER         (elementary renaming / permutation)
              Σ: ER → SIG             (signature implicit in renaming)
              ·: ER x SIG → SIG       (application of renaming)
              ·: ER x PM → PM         (application of renaming)
              ·: ER x PN → PN         (application of renaming)
    functions (new)
              ∂: PM → PM              (process removal)
              < = >: PN x PM → PM     (process declaration module)
end signature Σ_BMACP(A, N).
```

4. (META)VARIABLES, IMPLICIT EMBEDDING CONVENTION, AXIOM SCHEMES

4.1. Declaration of variables

The axioms of BMACP(A,γ,N) are introduced in the following section 5. There are some preliminaries to the listing of the axioms that need detailed attention. First we will declare the variables that will be used in the axioms:

```
begin variables for ΣBMACP(A, N)
     P0, Q0, R0 → PN        [metavariables over N]
     P, Q, R → PN
     X, Y, Z → PM
     U, V, W → SIG
     a0, b0, → AA           [metavariables over A]
     a, b, c → AP
     r → ER
end variables
```

We will discuss the meaning of the attribute metavariable below. First we need an explanation of the (partial) implicit embedding convention.

4.2. Implicit embedding convention, implicit abbreviation convention, generalised implicit embedding and abbreviation convention

The *implicit embedding convention* allows to omit the embedding functions i_aa_ap, i_ap_pm, i_pn_sig, i_pn_pm whenever this will not lead to ambiguities. This means that equations and terms will be written in such a way that it is always clear how to augment an expression with the embedding functions in order to obtain a well-typed term or equation. Thus a precondition for an expression, equation or conditional equation to be well-formed is that it allows a unique type assignment involving the introduction of embeddings. Here it is understood that a variable must always be declared with a unique type in advance.

The *implicit abbreviation convention* allows to use prefixes of the names of the embeddings if these contain enough information to obtain unambiguous typing. Thus besides i_aa_ap, i_ap_pm, i_pn_sig and i_pn_pm, also i, i_aa, i_ap and i_pn are admitted. The convention will the allow to complete the name of the embedding function such as to obtain a term that is well-typed. Again there must be a unique way to do this. Moreover it is allowed to add additional embeddings if that helps to obtain a well-formed expression or equation again under the condition that this can be done in a unique way only.

A peculiar fact is that the implicit embedding convention may weaken the language because some identities can only be written in a way that allows additional but unintended typings. By prefixing the equation with [¬IE] it is indicated that no implicit embeddings and abbreviations are used or allowed. This introduces a complication for the syntax which is completely harmless for

the human user. We provide two examples: (i) On basis of the signature $\Sigma_{BMACP}(A, N)$ one obtains the PM expression

$$T(i_pn_sig(P) + \Sigma(X)) + i_ap_pm(a) \cdot \delta + i_pn_pm(Q) \square (i_pn_pm(P) \| \langle P = i_aa_pm(b) \cdot i_pn_pm(P) + i_aa_pn(c)\rangle).$$

Using the implicit abbreviation convention this expression is written as:

$$T(i(P) + \Sigma(X)) + i(a) \cdot \delta + i(Q) \square (i(P) \| \langle P = i(b) \cdot i(P) + i(c)\rangle).$$

Then using the implicit embedding convention this expression is written as:

$$T(P + \Sigma(X)) + a \cdot \delta + Q \square (P \| \langle P = b \cdot P + c\rangle).$$

(ii) The equation $Id \cdot P = P$ is not well-formed because it has three different correct completions:

$Id \cdot P = P$, $Id \cdot i_pn_sig(P) = i_pn_sig(P)$ and

$Id \cdot i_pn_pm(P) = i_pn_pm(P)$.

Finally the ***generalised implicit embedding and abbreviation convention*** allows to use expressions having different completions to a well-typed expression if on basis of the other axioms the different completions can be proved equal. This convention should be used with care. A typical example is the expression $(r \cdot P) \| Q$. Obviously the subexpression $r \cdot P$ must have type PM. But that can be done using different completions: $r \cdot i_pn_pm(P)$ and $i_pn_pm(r \cdot P)$. Based on the presence of an axiom that identifies both expressions it is legitimate to use $r \cdot P$ as an expression of type PM.

It should be noticed that the mentioned conventions serve only one purpose: to allow a readable presentation of an algebraic specification in a many-sorted Language. At the level of abstract syntax one always means the fully disambiguated version of the text.

4.3. Axiom schemes with metavariables

The axioms involving variables over AA and PN should in fact be understood as axiom schemes where every possible substitution from A resp. N is generated. Therefore negative conditions such as $P \neq Q$ are allowed and must be understood as restrictions on the number of substitutions that are required. Thus with $A = \{a1, a2,.., ak\}$ and $N = \{p1, p2,..\}$ axioms should be read as follows:

$a \mid \delta = \delta$ stands for:

$$i_ap_pn(a) \mid i_ap_pn(\delta) = i_ap_pn(\delta)$$

$a0 \mid b0 = \gamma(a0, b0)$ if $G(a0, b0)$ is defined stands for

$$\{i_ap_pn(an) \mid i_ap_pn(am) = i_ap_pn(\gamma(an, am) \mid 1 \leq n \leq k,\ 1 \leq m \leq k,\ \gamma(an, bm) \text{ is defined}\}.$$

$\Sigma(r(P, Q)) = P + Q$ stands for

$$\Sigma(r(pn, pm)) = i_pn_sig(pn) + i_pn_sig(pm)$$

$[\neg IE]$ $r(P0, Q0) \cdot R0 = R0$ if $R0 \neq P0$ and $R0 \neq Q0$ stands for
$\{[\neg IE]\ r(pn, pm) \cdot pt = pt \mid 1 \leq n,\ 1 \leq m,\ n \neq t \neq m\}$

5 BMACP(A,γ,N), THE AXIOMS

5.1. The first two axioms determine the effect of the communication function on atoms.

[0] $a0 \mid b0 = \gamma(a0, b0)$ if $\gamma(a0, b0)$ is defined
[1] $a0 \mid b0 = \delta$ if $\gamma(a0, b0)$ is undefined

5.2. The axioms of ACP from [BK 84] minus the axiom that says that δ is a left zero for multiplication, which is modified into [8].

[2] $X + Y = Y + X$
[3] $(X + Y) + Z = X + (Y + Z)$
[4] $X + X = X$
[5] $(X + Y) \cdot Z = X \cdot Z + Y \cdot Z$
[6] $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$
[7] $\delta + X = X$
[8] $\delta \cdot X = \partial(X)$
[9] $a \mid b = b \mid a$
[10] $(a \mid b) \mid c = a \mid (b \mid c)$
[11] $\delta \mid a = \delta$
[12] $X \parallel Y = X \mathbin{|\!\lfloor} Y + Y \mathbin{|\!\lfloor} X + X \mid Y$
[13] $(a \cdot X) \mathbin{|\!\lfloor} Y = a \cdot (X \parallel Y)$
[14] $a \mathbin{|\!\lfloor} Y = a \cdot Y$
[15] $(X + Y) \mathbin{|\!\lfloor} Z = X \mathbin{|\!\lfloor} Z + Y \mathbin{|\!\lfloor} Z$
[16] $(a \cdot X) \mid b = (a \mid b) \cdot X$
[17] $a \mid (b \cdot X) = (a \mid b) \cdot X$
[18] $(a \cdot X) \mid (b \cdot Y) = (a \mid b) \cdot (X \parallel Y)$
[19] $X \mid (Y + Z) = X \mid Y + X \mid Z$
[20] $(X + Y) \mid Z = X \mid Z + Y \mid Z$
[21] $\partial_H(\delta) = \delta$
[22] $\partial_H(a0) = a0$ if $a0 \notin H$
[23] $\partial_H(a0) = \delta$ if $a0 \in H$
[24] $\partial_H(X + Y) = \partial_H(X) + \partial_H(Y)$
[25] $\partial_H(X \cdot Y) = \partial_H(X) \cdot \partial_H(Y)$

In view of equation 8 we need additional axioms for process removal in connection with ACP. (In

the present context the identity $\partial(X) = \partial_A(X)$ holds. The introduction of process removal, suggested to us by F.Vaandrager, is more systematic however because it allows extensions of the calculus where encapsulation and process removal will diverge.)

[26] $\partial(a) = \delta$
[27] $\partial(X + Y) = \partial(X) + \partial(Y)$
[28] $\partial(X \cdot Y) = \partial(X) + \partial(Y)$
[29] $\partial(X \parallel Y) = \partial(X) + \partial(Y)$
[30] $\partial(X ⌊⌊ Y) = \partial(X) + \partial(Y)$
[31] $\partial(X \mid Y) = \partial(X) + \partial(Y)$
[32] $\partial(\partial_H(X)) = \partial(X)$
[33] $\partial(P) = T(P)$
[34] $\partial(T(V)) = T(V)$
[35] $\partial(\langle P = X\rangle) = \langle P = X\rangle$
[36] $\partial(V \square X) = V \square \partial(X)$

5.3. Axioms concerning the boolean algebra SIG.

[37] $\emptyset + U = U$
[38] $U + V = V + U$
[39] $(U + V) + W = U + (V + W)$
[40] $U + U = U$
[41] $\emptyset \cap U = \emptyset$
[42] $U \cap V = V \cap U$
[43] $(U \cap V) \cap W = U \cap (V \cap W)$
[44] $U \cap U = U$
[45] $P0 \cap Q0 = \emptyset$ if $P0 \neq Q0$
[46] $(U + V) \cap W = (U \cap W) + (V \cap W)$

5.4. General axioms on renamings and their effects.

[47] $r(P, Q) = r(Q, P)$
[48] $r(P, P) = Id$
[49] $[\neg IE]$ $r(P, Q) \cdot (P) = Q$
[50] $[\neg IE]$ $r(P0, Q0) \cdot (R0) = R0$ if $P0 \neq R0$ and $Q0 \neq R0$
[51] $\Sigma(r(P, Q)) = P + Q$
[52] $r \cdot i_pn_sig(P) = i_pn_sig(r \cdot P)$
[53] $r \cdot \emptyset = \emptyset$
[54] $r \cdot (U + V) = (r \cdot U) + (r \cdot V)$

Renaming commutes with embedding in PM:

[55] $r \cdot i_pn_pm(P) = i_pn_pm(r \cdot P)$

5.5. The axioms of module algebra from [BHK 86] with some simplifications allowed by the fact that the present concepts of both a signature and an atomic module are simpler than the general ones in [BHK 86] and with some additions due to the presence of process algebra operators.

[56] $\Sigma(X) = \Sigma(\partial(X))$

[57] $\Sigma(\langle P = X \rangle) = P + \Sigma(X)$

[58] $\Sigma(P) = P$

[59] $\Sigma(T(U)) = U$

[60] $\Sigma(X + Y) = \Sigma(X) + \Sigma(Y)$

[61] $\Sigma(U \square X) = U \cap \Sigma(X)$

[62] $\Sigma(r \cdot X) = r \cdot \Sigma(X)$

[63] $r \cdot \langle P = X \rangle = \langle r \cdot P = r \cdot X \rangle$

[64] $r \cdot T(U) = T(r \cdot U)$

[65] $r \cdot (X + Y) = (r \cdot X) + (r \cdot Y)$

[66] $r \cdot (U \square X) = (r \cdot U) \square (r \cdot X)$

[67] $r \cdot (r \cdot X) = X$

[68] $\Sigma(r) \cap \Sigma(X) = \emptyset \rightarrow r \cdot X = X$

[2] $X + Y = Y + X$

[3] $(X + Y) + Z = X + (Y + Z)$

[69] $T(U + V) = T(U) + T(V)$

[70] $X + T(\Sigma(X)) = X$

[71] $X + \partial(U \square X) = X$

[72] $\Sigma(X) \square X = X$

[73] $U \square (V \square X)) = (U \cap V) \square X$

[74] $U \square (T(V) + X) = T(U \cap V) + (U \square X)$

[75] $(\Sigma(X) \cap \Sigma(Y)) = U \rightarrow U \square (X + Y) = (U \square X) + (U \square Y)$

5.6. Additional axioms for renamings.

[76] $r \cdot a = a$

[77] $r \cdot (X \cdot Y) = (r \cdot X) \cdot (r \cdot Y)$

[78] $r \cdot (X \parallel Y) = (r \cdot X) \parallel (r \cdot Y)$

[79] $r \cdot (X \mathbin{|\!\lfloor} Y) = (r \cdot X) \mathbin{|\!\lfloor} (r \cdot Y)$

[80] $r \cdot (X \mid Y) = (r \cdot X) \mid (r \cdot Y)$

[81] $r \cdot \partial_H(X) = \partial_H(r \cdot X)$

5.7. Axioms that determine the effect of process algebra constructors on expressions involving declarations:

[82] $\partial_H(\partial(Z)) = \partial(Z)$

[83] $(X + \partial(Z)) \cdot Y = X \cdot Y + \partial(Z)$

[84] $(X + \partial(Z)) \parallel Y = X \parallel Y + \partial(Z)$

[85] $(X + \partial(Z)) \mathbin{\lfloor\!\lfloor} Y = X \mathbin{\lfloor\!\lfloor} Y + \partial(Z)$

[86] $(X + \partial(Z)) \mid Y = X \mid Y + \partial(Z)$

[87] $X \cdot (Y + \partial(Z)) = X \cdot Y + \partial(Z)$

[88] $X \parallel (Y + \partial(Z)) = X \parallel Y + \partial(Z)$

[89] $X \mathbin{\lfloor\!\lfloor} (Y + \partial(Z)) = X \mathbin{\lfloor\!\lfloor} Y + \partial(Z)$

[90] $X \mid (Y + \partial(Z)) = X \mid Y + \partial(Z)$

5.8. Axioms that allow to remove declarations and exports from declaration bodies:

[91] $\langle P = X + \partial(Z)\rangle = \langle P = X\rangle + \partial(Z)$

[92] $P \cap \Sigma(X) = \emptyset \rightarrow \langle P = U \Box X\rangle = (U + P) \Box \langle P = X\rangle)$

[93] $P \cap U = P \rightarrow \langle P = U \Box X\rangle = U \Box \langle P = X\rangle$

5.9. Axioms that allow to commute export operators and process algebra operators. There are 4 conditional axioms with the same precondition.

[94] $\Sigma(X) \cap \Sigma(Y) = V \rightarrow V \Box (X \cdot Y) = (V \Box X) \cdot (V \Box Y)$

[95] $\Sigma(X) \cap \Sigma(Y) = V \rightarrow V \Box (X \parallel Y) = (V \Box X) \parallel (V \Box Y)$

[96] $\Sigma(X) \cap \Sigma(Y) = V \rightarrow V \Box (X \mathbin{\lfloor\!\lfloor} Y) = (V \Box X) \mathbin{\lfloor\!\lfloor} (V \Box Y)$

[97] $\Sigma(X) \cap \Sigma(Y) = V \rightarrow V \Box (X \mid Y) = (V \Box X) \mid (V \Box Y)$

[98] $\partial_H(V \Box X) = V \Box \partial_H(X)$

5.10. The redundant definition removal axiom.

[99] $P \cap (U + \Sigma(X) + \Sigma(Y)) = \emptyset \rightarrow U \Box (X + \langle P = Y\rangle) = U \Box (X + T(P) + \partial(Y))$

5.11. The body replacement axiom.

[100] $P + \langle P = X\rangle = X + \langle P = X\rangle$.

This completes the description of the axioms of BMACP. If a full specification is to be made in, for instance, ASF [BHK 89] the parameter sets A and N with equality function as well as the power set of A with uniform characteristic function ($\in$) must be specified in advance. This will not

generate serious difficulties. It is currently not clear to us how BMACP can be translated into a term rewiting system complete modulo some permutative reductions. That step is needed if it is to be specified in ASF as an executable algebraic specification. Moreover our description of the specification is rather unmodularised itself, and a coding of it in ASF should address that aspect as well.

6. NORMAL FORMS

6.1. Normal form theorem

NORMAL FORM THEOREM. *Let* M *be a closed module expression, i.e. a closed expression of sort* PM *over* $\Sigma_{BMACP}(A, N)$. *Then there exists a closed module expression* M' *which is provably equal to* M *in* BMACP(A, γ, N) *with the following form:*

$$M' = U \sqcap (M_0 + \langle P_1 = M_1\rangle + ... + \langle P_n = M_n\rangle + T(P_{n+1} + ... + P_{n+m}))$$

where the M_i *are process expressions over* $\Sigma_{ACP}(A, N)$ *and the names in* $T(P_{n+1} + ... + P_{n+m})$ *occur in* U *but not in the other summands. (Notice that for* $i, j \leq n$ P_i *and* P_j *may coincide.)*

PROOF. We provide a sketch only. Let us call a name occurrence of P hidden if it occurs within the scope of an operator $U \sqcap (.)$ in such a way that P is not contained in U. Further two occurrences P' and P'' of the name P have equal scope if every subterm M^* of M in which P' occurs in a hidden way also contains P''. Starting from M the first step is to rename all hidden occurrences of names in such a way that whenever a name has two occurrences these have equal scope. As an example of this procedure consider:

$$P \sqcap (d + \langle Q = a \cdot P + b\rangle) + c \cdot Q.$$

In this expression both occurrences of the name Q fail to have equal scope. Then choose an entirely new name, say R. Now $P \cap \Sigma(r(Q, R)) = \emptyset$ hence:

$$P \sqcap (d + \langle Q = a \cdot P + b\rangle) = r(Q, R) \cdot (P \sqcap (d + \langle Q = a \cdot P + b\rangle)) =$$

$$(r(Q, R) \cdot P) \sqcap (r(Q, R) \cdot (d + \langle Q = a \cdot P + b\rangle)) = P \sqcap (d + \langle R = a \cdot P + b\rangle).$$

It follows that

$$P \sqcap (d + \langle Q = a \cdot P + b\rangle) + c \cdot Q = P \sqcap (d + \langle R = a \cdot P + b\rangle) + c \cdot Q$$

which satisfies the requirements.

Once a term has been obtained in which all occurrences of process names have equal scope it follows that all export operators can be moved to an outermost position and then combined into a single application of the export operator. Thereafter the declarations have to be moved towards the roots of expressions. This is done as follows: on basis of the equations of BMACP it is allowed to replace $C[\langle P = K\rangle]$ by $C[\delta] + \langle P = K\rangle$. Of course this step presupposes the absence of export operators in the context $C[\ ]$. After finitely many of such steps a normal form is obtained.

6.2. A meaning for closed module expressions

Let $K = K(\Sigma_{ACP}(A))$ be a process algebra that has atomic actions A and a communication mechanism corresponding to γ. Suppose that K satisfies the axioms of ACP. Let π be a name not in N. The meaning of a closed module expression in normal form M will be given as a pair $SEM(K, M) = (\Sigma(M), F)$ of $\Sigma(M)$ and an element of $P(\Sigma(M) \cup \{\pi\} \to K)$. This is the powerset of the class of valuations from $\Sigma(M) \cup \{\pi\}$ to K. In other words F is a relation with attributes in $\Sigma(M) \cup \{\pi\}$ and with for each attribute the domain equal to $Dom(K)$. Let $M = U \square (M_0 + \langle P_1 = M_1\rangle + ... + \langle P_n = M_n\rangle + T(P_{n+1} + .. + P_{n+m}))$. The first component of $SEM(K, M)$ is just the visible signature of M. For the second component we first consider $M^* = M_0 + \langle P_1 = M_1\rangle + ... + \langle P_n = M_n\rangle + T(P_{n+1} + .. + P_{n+m})$. The meaning of M^* is the pair $(\Sigma(M^*), F^*)$ with F^* the collection of all valuations σ from $\Sigma(M^*) \cup \{\pi\}$ to K such that

$K, \sigma \models \langle P_1 = M_1\rangle + ... + \langle P_n = M_n\rangle$ and $(K, \sigma \models M_0) = \sigma(\pi)$.

F is then obtained from F^* by projection to $\Sigma(M) \cup \{\pi\} = U \cap \Sigma(M^*) \cup \{\pi\}$. We say that M has a well-defined semantics (in the terminology of 1.2: $M \in NMF^*$) if in F the attribute π is functionally dependent on the other atributes in $\Sigma(M)$. In this case we say that $SEM(K, M)$ is functional (of course with respect to K).

(This meaning function has its drawbacks. Another possibility is: $SEM^*(K, M) = U\{SEM(K, M') \mid M'$ a normal form congruent with $M\}$.)

6.3. Nice normal forms

It appears that the kind of 'nicety' one may define depends on the process algebra K. We consider the simplest case: $K = A_n$ the finite projection algebra with depth n. At this point we need the projection operators π_n for natural numbers $n > 0$. Defining equations for the projection operators are as follows (the projection axioms PR):

$$\pi_0(X) = \delta, \qquad \pi_n(X + Y) = \pi_n(X) + \pi_n(Y)$$
$$\pi_{n+1}(a) = a, \qquad \pi_{n+1}(a \cdot X) = a \cdot \pi_n(X)$$

We call the normal form $M = U \square (M_0 + \langle P_1 = M_1\rangle + ... + \langle P_n = M_n\rangle + T(P_{n+1} + ... + P_{n+m}))$ A_k*-nice* if for every valuation σ in $SEM(A_n, M)$ and for all finite terms $H_1, .. H_{n+m}$, H over $BPA_\delta(A)$ such that for all i: $1 \leq i \leq n+m$, $A_k, \sigma \models P_i = H_i$, and $A_k, \sigma \models \pi = H$ the following formal proof exists:

$$ACP(A, \gamma) + PR + \{\pi_k(X) = X\} + \{P_i = H_i \mid 1 \leq i \leq n+m\}$$
$$\vdash M_0 = H.$$

Concerning the model A_∞ the following definition is plausible: a normal form A_∞-nice if it is A_k -nice for every $k > 1$.

Notice that an A_n -nice specification may contain many redundant equations for instance: $\langle P = a \cdot P\rangle = \langle P = a \cdot P\rangle + \langle P = a \cdot a \cdot P\rangle$. That 'niceness' of a normal form is a non-trivial property related to guardedness of specifications can be understood from the following example. Let $A = \{a, b, c\}$. Then $\emptyset \square a \cdot P = (\emptyset \square a \cdot P) + (\emptyset \square a \cdot P) = (\emptyset \square a \cdot P) + (\emptyset \square a \cdot Q) = \emptyset \square (a \cdot P + a \cdot Q)$. Now $SEM(A_2, \emptyset \square (a \cdot P + a \cdot Q)) \neq SEM(A_2, \emptyset \square a \cdot P)$ because $a \cdot b + a \cdot c$ is in the one and not in the other.

6.4. Soundness theorem

SOUNDNESS THEOREM. *Let* $M1$ *and* $M2$ *be two normal forms. Moreover assume that* $BMACP(A, \gamma, N) \vdash M1 = M2$. *Then* $M1$ *is* A_n*-nice if and only* $M2$ *is* A_n*-nice and if both are* A_n*-nice then* $SEM(A_n, M1) = SEM(A_n, M2)$.

PROOF. Omitted.

REMARKS: (i) One really needs a mechanically verified proof of soundness because when put in use in practice it should be reproved for many extended calculi. Notice that from a methodological point of view one has to consider the possibility that the soundness theorem is wrong or a soundness theorem of useful form cannot be found. If so, what must be done? There are several possibilities.

(i) Correct minor mistakes in the axiom system of BMACP or an extension.
(ii) Restrict the notion of a nice normal form.
(iii) Work relative to another model of process algebra.
(iv) Uncouple module algebra combination and process algebra's alternative composition.

It is not clear to us how long these remedies will work. Indeed more general forms of BMACP result if one or more of the following features are taken into account as well: declarations of atomic actions, declarations of local data types, state operators, process creation, interrupts, signals, τ-abstraction, ε, generalised sums, generalised merges, specific scope rules for input data and temporal logic specifications. The real test for our proposal is that it survives these extensions.

(ii) This version of soundness is based on the interpretation of normal forms in the projective limit model. It would be nicer to have an interpretation based on an arbitrary process algebra. We expect, however, that a variety of different interpretations of BMACP can be found.

(iii) A typical example of the problems with the soundness theorem for BMACP is the following: $\emptyset \square (a \cdot P) = \emptyset \square (a \cdot P) + \emptyset \square (a \cdot P) = \emptyset \square (a \cdot P + a \cdot Q)$. The first and third of expressions have different semantics in the sense of 6.2. The difficulty can be remedied in at least

three ways:

(1) restrict attention to cases where all hidden process names are defined by means of guarded recursion equations and use the fact that these have unique solutions to ensure the validity of expressions of the form $SEM(X) = SEM(Y)$,

(2) use a mechanism of origin tracing like the origin consistency rule of ASF [BHK 89] and COLD [FJKR 87] in order to force renamed versions of the same hidden name to have the same meaning after normalisation,

(3) remove axioms that may introduce duplication of process expressions containing export operators (this motivates the modified axiom system BMACP* below).

7. A WEAKER SYSTEM BMACP*

We describe an axiom system BMACP* that is obtained from BMACP by applying three modifications. The relevant properties of this system are mentioned below.

(i) Restrict all axioms that introduce the multiplication of a PM-variable by replacing that variable by a variable over process names.

[4*]	$P + P = P$
[5*]	$(X + Y) \cdot P = X \cdot P + Y \cdot P$
[12*]	$P \parallel Q = P \mathbin{\|\!\lfloor} Q + Q \mathbin{\|\!\lfloor} P + P \mid Q$
[15*]	$(X + Y) \mathbin{\|\!\lfloor} P = X \mathbin{\|\!\lfloor} P + Y \mathbin{\|\!\lfloor} P$
[19*]	$P \mid (Y + Z) = P \mid Y + P \mid Z$
[20*]	$(X + Y) \mid P = X \mid P + Y \mid P$

(ii) Remove axiom [100] and add the inverse body replacement axiom:

[100*]	$P \cap \Sigma(X) = \emptyset \rightarrow X = \Sigma(X) \square (P + \langle P = X \rangle)$

(iii) Add the weak body replacement axioms:

[101]	$P + \langle P = a \rangle = a + \langle P = a \rangle$
[102]	$P + \langle P = Q + R \rangle = Q + R + \langle P = Q + R \rangle$
[103]	$P + \langle P = Q \cdot R \rangle = Q \cdot R + \langle P = Q \cdot R \rangle$
[104]	$P + \langle P = Q \parallel R \rangle = Q \parallel R + \langle P = Q \parallel R \rangle$
[105]	$P + \langle P = Q \mathbin{\|\!\lfloor} R \rangle = Q \mathbin{\|\!\lfloor} R + \langle P = Q \mathbin{\|\!\lfloor} R \rangle$
[106]	$P + \langle P = Q \mid R \rangle = Q \mid R + \langle P = Q \mid R \rangle$
[107]	$P + \langle P = \partial_H(Q) \rangle = \partial_H(Q) + \langle P = \partial_H(Q) \rangle$

The objective of BMACP* is to find an axiom system which is weaker than BMACP and still allows a normal form theorem of the required kind and allows to write process expressions (occurring in a normal form) into head normal form in all cases in which this is possible in

BMACP. Moreover BMACP* fails to feature axioms that duplicate (or even triplicate) process variables, in particular the axioms 4, 5, 12, 15, 19 and 20. Due to the absence of these axioms the semantic problems mentioned in 6 (iii) will not occur in connection with BMACP*.

8. EXAMPLES AND REMARKS

8.1. A derivation: $\langle P = X\rangle ⫾\!\!\!\lfloor (Q + \langle Q = a\rangle) = (\delta + X\rangle) ⫾\!\!\!\lfloor (Q + \langle Q = a\rangle) =$
$\delta ⫾\!\!\!\lfloor (Q + \langle Q = a\rangle) + \langle P = X\rangle = \delta \cdot (Q + \langle Q = a\rangle) + \langle P = X\rangle =$
$\partial_A(Q + \langle Q = a\rangle) + \langle P = X\rangle = \partial_A(a + \langle Q = a\rangle) + \langle P = X\rangle =$
$\partial_A(a) + \partial_A(\langle Q = a\rangle) + \langle P = X\rangle = \langle Q = a\rangle + \langle P = X\rangle$.

8.2. Consider the following process $X = a \cdot b^{\omega}$. It can be shown that if one intends to specify this process using guarded recursive equations over the signature of BPA at least two equations are needed, for instance $\langle P = a \cdot Q\rangle + \langle Q = b \cdot Q\rangle$. Using the μ construct this can be done with only one equation $\langle P = a \cdot (\mu Q.\ b \cdot Q)\rangle$. So the style of recursive specification by means of guarded recursion equations leads to more equations than seems necessary. In the formalism of BMACP the following specification can be given. $\langle P = a \cdot (Q + \langle Q = b \cdot Q\rangle)\rangle$. It is useful to introduce the notation $P := X$ as an abbreviation for $P + \langle P = X\rangle$. We call this the definition construct. Examples: $\langle P = a \cdot (Q := b \cdot Q)\rangle$, $a \cdot (Q := b \cdot Q)$. The difference between the definition construct and the usual fixed point operator $\mathrm{fixp}(F(P))$ or $\mu P.\ (F(P))$ is the large scope in which the name P is known. In the definition construct this scope is large in the sense that an explicit hiding (non-export) of the name is needed to restrict the scope. The fix (μ) construct allows α-conversion and does not show the name of the bound variable at all. A simulation of the μ-construct in BMACP can be given as follows: $a \cdot (\mu P.\ b \cdot P) = a \cdot \emptyset \,\square\, (P := b \cdot P)$.

8.3 The following derivable identities are useful:

(i) $\delta = T(\emptyset)$

Proof. $\delta = \delta + T(\Sigma(\delta)) = \delta + T(\emptyset) = T(\emptyset)$.

(ii) if $p \cap \Sigma(X) = \emptyset$ then $X = \Sigma(X) \,\square\, (P + \langle P = X\rangle)$

Proof. $X =_{72} \Sigma(X) \,\square\, X =_{74\text{(and (1) above)}} \Sigma(X) \,\square\, (X + T(P)) =_{71\text{(with } U = \Sigma(X))} \Sigma(X) \,\square\, (X + T(P) + \partial(X)) =_{99} \Sigma(X) \,\square\, (X + \langle P = X\rangle) =_{100} \Sigma(X) \,\square\, (P + \langle P = X\rangle)$

8.4. An operational semantics for BMACP is given with the following 2 rules on top of the axioms: $a + X \rightarrow^{a} \surd$, $\quad a \cdot X + Y \rightarrow^{a} X + \partial(Y)$.

In order to apply these rules declarations must, when possible, be moved inside the part of a process expression that will not be erased.

8.5. Of course the system BMACP as such is of no practical value. The semantic problem involved in scope control of process names will not occur in isolation. A similar method may however be applied in more complicated circumstances. Then it may be a useful simplification to remove the axioms of process algebra that describe dynamic behavior and the axioms of module algebra that are not essential for normalisation altogether. For instance removing axioms 4, 5, 7 to 23 but introducing associativity and commutativity of merge yields a system BMACP' for which a more appealing form of soundness can be proven.

8.6. The signature of BMACP can be used as an abstract syntax for a specification language. The axioms of BMACP allow 'correct' transformations of this abstract syntax. The need for such transformations (but admittedly not for all transformations of BMACP) can be motivated as follows. An appropriate way to design a specification language for ACP is to simultaneously design an abstract syntax, a vertical syntax with key words etc., a graphical/object oriented syntax and a horizontal (mathematical) syntax.

For the abstract syntax one may use an algebraic format with prefix notation. For the horizontal syntax one will replace many prefix operators by infix and mixfix operators, use mathematical symbols and Greek characters, introduce priorities, implicit type conversion and so on. The vertical syntax will allow the explicit use of type and structure information in headers and footers of sections. The graphical syntax will remove the redundant sequential order information that is unavoidable with both horizontal and vertical representations and allow a very flexible way of presenting type and structure information in both a static and a dynamic way. (Notice that our current presentation of BMACP yields a mixture of an abstract and a horizontal syntax. It can be transformed to an abstract syntax by writing all axioms in prefix notation and removing the implicit embedding convention. It can be turned into a horizontal syntax by adding precise information about priorities, bracketing and implicit embeddings.)

Then it must be defined what it means for two representations, say a graphical representation G and a vertical representation V to 'represent the same specification' i.e. whether or not G and H are 'equivalent'. We propose that this question is to be settled via the abstract syntax. Both representations are first transformed to an abstract representation in a uniform way to be defined in advance. This leads to abs(G) and abs(V). Then G and H are considered equivalent if BMACP proves abs(G) and abs(V) equal.

8.7. Acknowledgements. Frits Vaandrager and Rob van Glabbeek have contributed substantially in the final design of the axiom system.

9. REFERENCES

[deBZ 82] J.W. DE BAKKER & J.I. ZUCKER, *Processes and the denotational semantics of concurrency*, Information & Control 54 (1/2), (1982) 153-158

[BHK 86] J.A. BERGSTRA, J. HEERING & P. KLINT, *Module algebra*, Report CS-R8617, Centre for Mathematics and Computer Science, Amsterdam (1986)

[BHK 89] J.A. BERGSTRA, J. HEERING & P. KLINT, (eds.) *Algebraic Specification*, Addison Wesley, ACM Press Frontier Series (1989)

[BK 82] J.A. BERGSTRA & J.W. KLOP, *Fixed point semantics in process algebra*, Mathematical Centre Research Report, IW 206 (1982)

[BK 84] J.A. BERGSTRA & J.W. KLOP, *Process algebra for synchronous communication*, Information & Control 60 (1/3), (1984) 109-137

[B 88] E. BRINKSMA, *On the design of extended LOTOS, A specification language for distributed open systems*, Ph.D.Thesis, University of Twente (1988)

[FJKR 87] L.G.M. FEYS, H.B.M. JONKERS, C.J.P. KOYMANS & G.R. RENARDEL DE LAVALETTE, *Formal definition of the design language* COLD-K, Technical Report, ESPRIT Project 432 METEOR, Philips Research Eindhoven (1987)

[vGV 88] R.J. VAN GLABBEEK & F.W. VAANDRAGER, *Modular specifications in process algebra—with curious queues*, Centre for Mathematics and Computer Science, Report CS-R8821 (1988).

[H 78] C.A.R. HOARE, *Communicating sequential processes*, Comm.ACM 21 (1978) 666-677

[MV 88] S. MAUW & G.J. VELTINK, *A process specification formalism*, University of Amsterdam, Programming Research Group, Report P8814 (1988)

[M 80] R. MILNER, *A calculus of communicating systems*, Springer LNCS, 92 (1980)

[SP 89] SPECS-Consortium / PTT-RNL, *Definition of MR, Version 1*, SPECS document D.WP5.2 (1989)

Towards Compositional Predicate Transformer Semantics for Concurrent Programs

Eike Best[1]
Institut für Informatik
Universität Hildesheim
D - 3200 Hildesheim

Written on the occasion of the 25th anniversary of Jaco de Bakker's involvement with the Mathematisch Centrum – Centrum voor Wiskunde en Informatica, Amsterdam.

Abstract

For a simple concurrent programming language, a slight modification of the formal framework exposed in [2] is shown to yield a compositional predicate transformer semantics which is consistent with denotational semantics.

1 Introduction

The seminal paper [2] exposes a denotational framework for the definition of the semantics of concurrent programs. We shall explore an idea, due to Jaco de Bakker, to modify this environment to accommodate predicate transformer semantics. For a simple concurrent programming language, we shall show that this modification leads to a generalisation of the well known consistency between relational ('forward') and predicate transformer ('backward') semantics of sequential programs [1].

In section 2 we briefly recall the necessary definitions for sequential programs. Section 3 contains the definitions of forward and backward semantics for simple concurrent programs, along with an example. Section 4 states (and partly proves) the consistency between them. Section 5 contains conclusions.

[1]This work was done while the author was with the Gesellschaft für Mathematik und Datenverarbeitung, D-5205 Sankt Augustin.

2 Sequential programs

Let a be a sequential (possibly nondeterministic) program and let S be the state space of a. The relational semantics $r(a) \subseteq S \times S$ defines $(s, s') \in r(a)$ iff starting with the state s, there is an execution of a terminating in the state s'. The predicate transformer[2] semantics $w(a): 2^S \rightarrow 2^S$ defines $w(a, X) = Y$ (for $X, Y \subseteq S$) iff Y is the set of initial states s such that starting with s, every terminating execution of a leads to a final state in X.
Between $r(a)$ and $w(a)$ there is the following relationship [1]:

$$\forall s \in S \, \forall X \subseteq S: s \in w(a, X) \iff r(a, s) \subseteq X. \tag{1}$$

(Informal justification: $s \in w(a, X)$ iff every terminating execution leads into X iff $r(a, s) \subseteq X$.)
For the purpose of explaining our example below, we need to define the relation r and the function w for simple sequential programs of the form $B \rightarrow x := e$, that is, assignments guarded by Boolean expressions. We define $(s, s') \in r(B \rightarrow x := e)$ iff $B(s) = \textbf{true}$ and s' equals s_x^e, where s_x^e is the same state as s except that the value of x in s_x^e equals e (as evaluated in s). Moreover, we define $Y = w(B \rightarrow x := e, X)$ iff for all $s \in Y$, $B(s) = \textbf{true}$ implies that $s_x^e \in X$. Unguarded assignments $x := e$ are a special case (consider $\textbf{true} \rightarrow x := e$). Plain Boolean expressions B are special cases as well (consider $B \rightarrow \textbf{skip}$).

3 Simple concurrent programs

Let c denote a shared variable program containing atomic actions as primitives and choice, sequence and parallel composition as combinators. We assume the following syntax for c:

$$c \quad ::= \quad a \mid c_1 \,\square\, c_2 \mid c_1 ; c_2 \mid c_1 \| c_2,$$

where a denotes an atomic action of the general form $\langle B \rightarrow x := e \rangle$. Let S denote the state space of c. For an atomic action a, the relation $r(a) \subseteq S \times S$ and the function $w(a): 2^S \rightarrow 2^S$ are defined as in section 2.
As in [2], an object representing the denotation of c will be defined as an element of a domain $\mathcal{P}$ satisfying the following domain equation:

$$\mathcal{P} \quad = \quad \{p_0\} \cup 2_c^{((S \times S) \times \mathcal{P})} \tag{2}$$

where 2_c is the set of all closed subsets (see [2] for the definition of closedness).

[2]Instead of predicates over S we shall consider subsets of S, i.e., elements of 2^S.

Remark:

In [2], the following domain equation has been used instead of equation (2):

$$\mathcal{P}' \quad = \quad \{p'_0\} \cup (S \to 2_c^{(S \times \mathcal{P}')}). \tag{3}$$

However objects $p \in \mathcal{P}$ can be translated equivalently into objects $p' \in \mathcal{P}'$ and vice versa.

To translate p into p', p_0 is translated into p'_0 by definition. Let $p = \{(r_1, p_1), \ldots, (r_m, p_m)\} \in \mathcal{P}$ with $p_j \in \mathcal{P}$; then p is translated into $p' = \lambda s.X$ with $(s_i, p'_i) \in X$ iff there is a pair $(r_j, p_j) \in p$ such that $(s, s_i) \in r_j$ and p_j is translated into p'_i.

Conversely, to translate p' into p, p'_0 is translated into p_0 by definition. Let $p' = \lambda s.\{(s_1, p'_1), \ldots, (s_n, p'_n)\} \in \mathcal{P}'$ with $p'_i \in \mathcal{P}'$. Then p' is translated into $p = \{(r_1, p_1), \ldots, (r_n, p_n)\} \in \mathcal{P}$ such that $r_i \subseteq S \times S$ is defined by means of $(s, s') \in r_i \iff (s', p_i) \in p'(s)$, and p'_i is translated into p_i.

These two constructions are inverses of each other [3]. Hence objects in $\mathcal{P}$ and objects in $\mathcal{P}'$ can be used interchangeably. For convenience, we shall use $\mathcal{P}$.

The following will be the defining equation for the semantic domain $\mathcal{Q}$ of the predicate transformer:

$$\mathcal{Q} \quad = \quad \{q_0\} \cup 2_c^{((2^S \to 2^S) \times \mathcal{Q})} \tag{4}$$

Let c be a concurrent program. We define its 'forward' denotation $[\![c\rangle$ and its 'backward' denotation $\langle c]\!]$ as follows:

$$\begin{array}{lcl|rcl}
[\![a\rangle & = & \{(r(a), p_0)\} & \langle a]\!] & = & \{(w(a), q_0)\} \\
[\![c_1 \,\Box\, c_2\rangle & = & [\![c_1\rangle \cup [\![c_2\rangle & \langle c_1 \,\Box\, c_2]\!] & = & \langle c_1]\!] \cup \langle c_2]\!] \\
[\![c_1; c_2\rangle & = & [\![c_1\rangle \circ [\![c_2\rangle & \langle c_1; c_2]\!] & = & \langle c_2]\!] \circ \langle c_1]\!] \\
[\![c_1 \| c_2\rangle & = & [\![c_1\rangle \| [\![c_2\rangle & \langle c_1 \| c_2]\!] & = & \langle c_1]\!] \| \langle c_2]\!],
\end{array}$$

where $\|$ is as defined in [2] and $\circ$ is the reverse of the operation defined there (for notational convenience).

As an example, consider the following program c_0:

$$\textbf{var } x\textbf{: integer;}$$
$$(\underbrace{\langle x := x+1\rangle}_{a}; (\underbrace{\langle x = 0 \to x := 3\rangle}_{b_1} \,\Box\, \underbrace{\langle x \neq 0\rangle}_{b_2})) \,\|\, \underbrace{\langle x := x+2\rangle}_{d}.$$

Then $[\![c_0\rangle$ is the following set:

$$\begin{aligned}[\![c_0\rangle \;=\; p_1 &= \{(r(a),p_2),(r(d),p_3)\}\\ p_2 &= \{(r(b_1),p_4),(r(b_2),p_4),(r(d),p_5)\}\\ p_3 &= \{(r(a),p_5)\}\\ p_4 &= \{(r(d),p_0)\}\\ p_5 &= \{(r(b_1),p_0),(r(b_2),p_0)\}.\end{aligned}$$

On the other hand, $\langle c_0]\!]$ is the following set:

$$\begin{aligned}\langle c_0]\!] \;=\; q_1 &= \{(w(b_1),q_2),(w(b_2),q_2),(w(d),q_3)\}\\ q_2 &= \{(w(a),q_4),(w(d),q_5)\}\\ q_3 &= \{(w(b_1),q_5),(w(b_2),q_5)\}\\ q_4 &= \{(w(d),q_0)\}\\ q_5 &= \{(w(a),q_0)\}.\end{aligned}$$

Both expressions can be viewed as labelled trees [2]; however, despite the close symmetry in the definition, these trees are not, in general, isomorphic. The objects $[\![c\rangle$ and $\langle c]\!]$ contain enough information for the relational semantics and the predicate transformer semantics of c to be easily derivable. To this end we define two semantic functions ρ and ϕ with the following functionality:

$$\begin{aligned}\rho&: \mathcal{P}\times S \to 2^S\\ \phi&: \mathcal{Q}\times 2^S \to 2^S.\end{aligned}$$

Let $p \in \mathcal{P}$ and $s \in S$. Then $\rho(p,s)$ is defined recursively as follows:

$$\begin{aligned}\rho(p_0,s) &= \{s\}\\ \rho(p,s) &= \bigcup\{\rho(p',t) \mid \exists r \subseteq S\times S\colon (r,p')\in p \wedge (s,t)\in r\}.\end{aligned} \tag{5}$$

Let $q \in \mathcal{Q}$ and $X \subseteq S$. Then $\phi(q,X)$ is defined recursively as follows:

$$\begin{aligned}\phi(q_0,X) &= X\\ \phi(q,X) &= \bigcap\{\phi(q',Y) \mid \exists w\colon 2^S\to 2^S\colon (w,q')\in q \wedge Y = w(X)\}.\end{aligned} \tag{6}$$

The intention is that $\rho(p,s)$ describes the set of final states that are reachable from s by execution paths through p. For a final state to be included in this set, it suffices that there is one path by which it is reachable; hence the union quantifier in formula (5). On the other hand, $\phi(q,X)$ describes the intersection of all sets of initial states that are reachable by paths through q. The intention is that this set describes the predicate transformer of X backward through p (where p is related to q as $[\![c\rangle$ is related to $\langle c]\!]$). For an initial state to be in it, all possible execution paths through p have to lead into X; hence the intersection quantifier in formula (6).

As an example, reconsider the program c_0 with $p_1 = [c\rangle$ and $q_1 = \langle c]$. Let s denote the initial state $x = -1$. For convenience, we will denote states $x = v$ simply by v (hence s denotes the state -1). We compute $\rho(p_1, -1)$:

$$\begin{aligned}
\rho(p_1, -1) &= \bigcup\{\rho(p_2, 0), \rho(p_3, 1)\} \\
&= \rho(p_2, 0) \cup \rho(p_3, 1) \\
\rho(p_2, 0) &= \rho(p_4, 3) \cup \rho(p_5, 2) \\
\rho(p_3, 1) &= \rho(p_5, 2) \\
\rho(p_4, 3) &= \rho(p_0, 5) = \{5\} \\
\rho(p_5, 2) &= \rho(p_0, 2) = \{2\}.
\end{aligned}$$

Hence $\rho(p_1, -1) = \{2, 5\}$.
Let X denote the set of final states $\{2,3,4,5\}$. We compute $\phi(q_1, X)$:

$$\begin{aligned}
\phi(q_1, \{2,3,4,5\}) &= \phi(q_2, S) \cap \phi(q_2, \{0,2,3,4,5\}) \cap \phi(q_3, \{0,1,2,3\}) \\
&= \phi(q_2, \{0,2,3,4,5\}) \cap \phi(q_3, \{0,1,2,3\}) \\
\phi(q_2, \{0,2,3,4,5\}) &= \phi(q_4, \{-1,1,2,3,4\}) \cap \phi(q_5, \{-2,0,1,2,3\}) \\
\phi(q_3, \{0,1,2,3\}) &= \phi(q_5, S) \cap \phi(q_5, \{0,1,2,3\}) \\
&= \phi(q_5, \{0,1,2,3\}) \\
\phi(q_4, \{-1,1,2,3,4\}) &= \phi(q_0, \{-3,-1,0,1,2\}) = \{-3,-1,0,1,2\} \\
\phi(q_5, \{0,1,2,3\}) &= \phi(q_0, \{-1,0,1,2\}) = \{-1,0,1,2\}.
\end{aligned}$$

Hence $\phi(q_1, \{2,3,4,5\}) = \{-1,0,1,2\}$, as could be expected.

4 Consistency

In this section we state (and partially prove) a generalisation of equation (1). Let c be a concurrent program and let S be its state space as in the previous section. Then we claim that the following holds:

$$\forall s \in S\; \forall X \subseteq S\colon\; \underbrace{s \in \phi(\langle c], X)}_{lhs} \iff \underbrace{\rho([c\rangle, s) \subseteq X}_{rhs}. \tag{7}$$

In the proof of (7), we may proceed by structural induction over the syntax of c. However, we deal only with the two cases $c = a$ and $c = c_1 \,\square\, c_2$.

Case 1: $c = a$.

$$\begin{aligned}
lhs &= s \in \phi(\langle c], X) && \\
&\iff s \in \phi(\{(w(a), q_0)\}, X) && \text{(Def. } \langle x]) \\
&\iff s \in \phi(q_0, w(a, X)) && \text{(Def. } \phi) \\
&\iff s \in w(a, X) && \text{(Def. } \phi) \\
rhs &= \rho([c\rangle, s) \subseteq X && \\
&\iff \rho(\{(r(a), p_0)\}, s) \subseteq X && \text{(Def. } [c\rangle) \\
&\iff (\bigcup\{\rho(p_0, t) \mid (s,t) \in r(a)\}) \subseteq X && \text{(Def. } \rho) \\
&\iff (\bigcup\{\{t\} \mid (s,t) \in r(a)\}) \subseteq X && \text{(Def. } \rho) \\
&\iff r(a, s) \subseteq X && \text{(rewriting).}
\end{aligned}$$

Now the claim that *lhs* $\iff$ *rhs* follows directly from equality (1).

Case 2: $c = c_1 \,\Box\, c_2$.

The proof is conducted in two steps. First, *lhs* and *rhs* are rewritten in a more convenient form amenable to induction. Then, their equality is shown using the induction hypothesis for c_1 and c_2.

$$
\begin{array}{lll}
lhs & = & s \in \phi(\langle c], X) \\
 & \iff & s \in \phi(\langle c_1] \cup \langle c_2], X) \qquad \text{(Def. } \langle c]) \\
 & \iff & s \in \bigcap\{\phi(q', Y) \mid \exists w: 2^S \to 2^S: \\
 & & \quad (w, q') \in (\langle c_1] \cup \langle c_2]) \wedge Y = w(X)\} \qquad \text{(Def. } \phi) \\
 & \iff & \forall q' \in \mathcal{Q}\, \forall Y \subseteq S: [\exists w: 2^S \to 2^S: \\
 & & \quad (w, q') \in \langle c_i] \wedge Y = w(X)] \Rightarrow s \in \phi(q', Y) \quad \text{(for } i = 1, 2)
\end{array}
$$

$$
\begin{array}{lll}
rhs & = & \rho([c\rangle, s) \subseteq X \\
 & \iff & X \supseteq \rho([c_1\rangle \cup [c_2\rangle, s) \qquad \text{(Def. } [c\rangle) \\
 & \iff & X \supseteq (\bigcup\{\rho(p', t) \mid \exists r \subseteq S \times S: \\
 & & \quad (r, p') \in ([c_1\rangle \cup [c_2\rangle) \wedge (s, t) \in r\}) \qquad \text{(Def. } \rho) \\
 & \iff & \forall p' \in \mathcal{P}\, \forall t \in S: [\exists r \subseteq S \times S: \\
 & & \quad (r, p') \in [c_i\rangle \wedge (s, t) \in r] \Rightarrow \rho(p', t) \subseteq X \quad \text{(for } i = 1, 2)
\end{array}
$$

Next we prove *lhs* $\Rightarrow$ *rhs*.

Consider any $q' \in \mathcal{Q}$ and $w: 2^S \to 2^S$ such that $(w, q') \in \langle c_i]$, and define $Y = w(X)$. Then by *lhs*, we have $s \in \phi(q', Y)$. Hence s is in the intersection of all sets $\phi(q', Y)$ such that q' and Y have the above properties and hence, by the definition of ϕ, we have $s \in \phi(\langle c_i], X)$. By induction hypothesis, $\rho([c_i\rangle, s) \subseteq X$ holds true. Hence by the definition of ρ, we have $\rho(p', t) \subseteq X$ whenever $p' \in \mathcal{P}$, $t \in S$ and $r \subseteq S \times S$ are such that $(r, p') \in [c_i\rangle$ and $(s, t) \in r$. But this means that *rhs* holds.

The proof of *rhs* $\Rightarrow$ *lhs* is similar.

5 Conclusions

When $[c\rangle$ and $\langle c]$ are viewed as labelled trees then their relationship can be characterised by 'tree inversion', an inverse tree to a given tree being determined by having the reverse edge walks, with appropriately matched edge labels. Although tree inversion in this sense is not unique, there exists an inductive definition for it [3]. By means of this inductive definition, it becomes possible to derive relational semantics and predicate transformer semantics from each other, rather than from the common underlying program.

It appears difficult to uphold the idea of tree inversion for iterative or recursive concurrent programs, because trees containing infinite paths cannot

be inverted. Hence this author's preferred approach is to suggest – syntactic or semantic – means of guaranteeing the termination of any iterative or recursive loops. This would lead to trees which may be infinitely broad but are always finitely long and can be inverted, quite in contrast to the case of sequential boundedly nondeterministic programs which lead to finitely broad (but possibly infinitely long) trees.

Acknowledgement

The work described in this short note was directly stimulated by Jaco de Bakker's questions and remarks. It was done during an enjoyable stay of the author in Amsterdam in October 1983. At that time we were not aware of any other work on predicate transformers for concurrent programs. In the meantime, and independently, at least two other papers have been published on the subject: [4] and [5].

References

[1] J.W. de Bakker: *Mathematical Theory of Program Correctness.* Prentice Hall (1980).

[2] J.W. de Bakker und J. Zucker: *Processes and the Denotational Semantics of Concurrency.* Information and Control, Vol.54, No.1/2, pp.70-120 (1982).

[3] E. Best: *An Equivalence Between Plotkin's Term Rewrite Semantics, Control Sequence Semantics, and the Process Semantics of de Bakker and Zucker.* BEGRUND-Memorandum No.33 (April 1984). *Towards a General Equivalence of 'Forward' Semantics and Predicate Transformer Semantics for Concurrent Programs.* BEGRUND-Memorandum No.34 (April 1984).

[4] T. Elrad and N. Francez: *A Weakest Precondition Semantics for Communicating Processes.* TCS 29, pp.231-250 (1984).

[5] L. Lamport: win *and* sin*: Predicate Transformers for Concurrency.* Research Report No.17, Digital Equipment Corporation, Systems Research Center (May 1987). (To appear in TOPLAS.)

Facets of Software Development

Computer Science & Programming, Engineering & Management

Liber Amicorum:

Professor Jaco W. de Bakker

Dines Bjørner
Department of Computer Science
Technical University of Denmark
DK-2800 Lyngby
Denmark

March 9, 1989

Abstract

The Mathematics Centrum — now the Center for Mathematics and Informatics (MC/CWI) — has contributed significantly to the computation sciences. The involvement of the CWI in large scale information technology projects appears to destine the CWI to also contribute to computation engineering.

In this note allow me to speculate on a context in which our many, individual contributions to the computation sciences may fit into practical life.

Contents

1 Introduction 1

2 Four Axes of Software Development 2
2.1 Definition of Disciplines 2
2.1.1 Computation Sciences and Engineering 2
2.1.2 Computer Science 3
2.1.3 Computing Science 3
2.1.4 Systems "Science" 3
2.1.5 Hardware and Software Development 3
2.2 Characterisation of Developer Rôles 4
2.2.1 Management and Managers 4
2.2.2 Engineering and Engineers 4
2.2.3 Programming and Programmers 5
2.2.4 Theory Checking and Computation Scientists 6
2.3 Programs and Software 6
2.4 The Four Axes 7

3 Software Quality Measures 7
3.1 Software Product Qualities . . . 7
3.2 Software Development Project Qualities . . . 10

4 Methods and Principles 11
4.1 Methods . . . 11
4.2 Principles . . . 12
4.3 Techniques and Tools . . . 13

5 Conclusion 13

1 Introduction

Three more-or-less un-co-ordinated directions of advanced engineering development and scientific research are taking place within computation today: (a) very large scale, advanced systems and software development environments (SDEs) are being "researched", built and experimented with — by the european information technology industry; (b) the European Economic Community (EEC) has, within the ordinary ESPRIT programme, involved a large number of computer scientists in a number of rather goal-oriented, ie. applied research projects; and (c) "good old computer science" — as pursued by individual scientists — is still adding stone-upon-stone of interesting results to a "mountain" of paper.

In relation to point (a), this note reflects upon (i) the professions and professionals who are going to use these environments, (ii) the project and product qualities these environments are to support, and (iii) the methods, principles and techniques according to which the professionals use the environments.

This note attempts to taxonomise the disciplines and professions of the field of software development. We do so by enumerating salient features of 'Management', 'Software Engineering', 'Programming', and 'Computer Science'. The harmonious co-operation of professionals of these fields is necessary in the successful development of successful software products. Such harmonious co-operation must be guided by a method, with its techniques and tools, clearly perceived by all professionals.

The note is organised as follows: in section 2 we define four software development professions and the tasks of four groups of corresponding professionals. In section 3 we outline the kind of project and product qualities that these professions and professionals must all strive to achieve. And, finally, in section 4 we briefly mention some issues of methodological nature — such which should bind the professionals together, and into whose moulds our engineering, programming and scientific research should deliver practically useful results.

2 Four Axes of Software Development

2.1 Definition of Disciplines

There is the subject field, and there are its practitioners. In this subsection (2.1) we deal with the former, while in the next subsection (2.2) we deal with the latter. The

last subsection (2.3) of this section then expands on the field and its practitioners.

2.1.1 Computation Sciences and Engineering

Definition 1 *The computation sciences deal with what objects (data and processes) can exist inside computers and how they are constructed.*

Characterisation 1 *The computation sciences consist of computer science, computing science and hardware+software "science"*[1].

The above characterisation, although presently left mostly undefined, is perhaps dogmatic, but it will work well for us; and whatever short-comings it might have, will not shine through too much.

Definition 2 *Computation engineering consists of computer (or hardware) development and software development.*

So we divide our world into science and engineering. In science we try to understand; in engineering we succeed in practice.

2.1.2 Computer Science

Definition 3 *Computer science is the mathematical study of programs.*

In computer science we investigate theories of programs: of **what** programs are, of classes of program schemes, of what can be computed, of computational complexity (how difficult it is to compute), of computational models, and of the mathematical tools and techniques necessary to express and further develop that study.

Examples of sub-studies within computer science are: automata theory, formal languages, program schemata, complexity theory, mathematics of computation (incl. recursive function theory and proof theory studies), computational geometry, theory of denotational semantics (Scott domains, metric speaces, power domains, etc.), theory of algebraic semantics, computational category theory, etc.

2.1.3 Computing Science

Definition 4 *Computing science is the mathematical study of programming.*

In computing science we study **how** to construct, or develop, programs. Another word for computing science is programming methodology.

Examples of sub-studies within computing science are: information systems analysis, (conceptual) data modelling, requirements definition, functionality definition (ie. architectural specification), program transformation (incl. stepwise program refinement) techniques, inductive and deductive program development, program correctness verification and proof, machine coding techniques, functional programming, logic programming, imperative programming, parallel programming, compiling techniques, database techniques, etc.

[1] We put double quotes around 'science' since we do think that — whatever it is — it is presently not a science.

2.1.4 Systems "Science"

Definition 5 *Systems science is the pragmatic, empirical study of hardware/software systems and their development.*

In systems "science" we study the practical aspects of development: modularisation — for purposes of re- and co-operative use, version control and configuration, variant journaling, test case generation and validation, requirements & design decision tracking, development management monitoring & control, resource estimation, process (ie. development management) models, as well as the pragmatics of tooling for these practical aspects.

2.1.5 Hardware and Software Development

Definition 6 *Hardware and software development consists of, or is the practice of management, engineering, programming and theory checking.*

These are then the four axes of software[2] development: its management, its engineering, its programming, and its theory checking.

o o o

Note 1 *The above definitions are just that. You may not agree with them. You may rather wish to prefer already established "definitions" of the named fields. Be that as it may. If you think you disagree, then replace, for example, 'computer science' and 'computing science' with x and y. Now you have less grounds for disagreement! So: please do not get hung up on our definitions. It is always useful to attempt to clarify definitions — if for nothing else, then as an analytic exercise that sharpens our insight.*

2.2 Characterisation of Developer Rôles

From now on we limit our attention to software.

The previous subsection postulated a quartet of sub-disciplines, and hence of potentially distinct developer rôles in software development. We shall now further characterise these.

2.2.1 Management and Managers

Characterisation 2 *The development manager deals with resources.*

Characterisation 3 *The development manager plans overall development, estimate resource requirements, budgets these, establishes financing for these, allocate and schedule resources (people, machine (and other tools), time), and monitors and controls resources.*

[2]— and, correspondingly, hardware

Characterisation 4 *The development manager builds the organisations required to carry out development.*

Characterisation 5 *Management builds on the laws of sociology (econometrics, management science, etc.).*

Characterisation 6 *Managers treat resources as social objects: humans, money, time.*

2.2.2 Engineering and Engineers

Characterisation 7 *The software engineer accomodates programs and programming to existing hard and soft technologies.*

Characterisation 8 *The software engineer deals with such issues as fitting development tools to existing computer systems, constructing versions of software, configuring products from versions, provide for journaling of variant developments, build test cases, validate designs and implementations, etc.*

Characterisation 9 *The software engineer builds tools.*

Characterisation 10 *Software engineering builds on laws of software "science" and the natural sciences.*

The laws of the natural science enter our concern in connection with physical (electrical and other) limitations of ("the always current") technology.

Characterisation 11 *The software engineer treats software as physical objects.*

Software is subject to execution, and execution behaviours are examined (say for purposes of testing and validation), related documents are "walked through", and in general: quality assurance is something which treats documents and code syntactically.

Note 2 *Our definition of 'Software Engineering', and its implications, appears to be different from that of 'Software Engineering' as generally formulated. We stick to our definition. We cannot make much sense out of any other definition. Still we are not entirely happy with the situation altogether.*

Note 3 *European computation sciences seem to have focused very much on contributions to computer and computing science, whereas US computation sciences appears to have contributed to not quite overlapping facets of computer science and to software engineering.*

2.2.3 Programming and Programmers

Characterisation 12 *The programmer defines requirements, designs the architectural components, specifies functionalities, transforms specifications into code, and argues correctness of these steps.*

Characterisation 13 *The programmer constructs theories by adhering to rules of programming methodology.*

Characterisation 14 *The programmer builds on laws of computer and computing sciences.*

Characterisation 15 *Programmers treat programs and programming as formal, mathematical objects.*

Programmers reason about the formal, mathematical objects denoted by what is written down in formal requirements, specification, design and code documents, and the relations between such documents.

2.2.4 Theory Checking and Computation Scientists

Programs express theories. Programming is "executing meta-programs". New software applications usually go beyond state-of-the-art understanding.

Characterisation 16 *The resident computation scientist checks whether the programmer expresses proper theories.*

Requirements definitions, functionality specification, and program transformation do not always, all the time, stay within established theories. Usually, however, what the programmer intends is perfectly permissible, only the theories appear to not cater for the special cases, and the resident scientists must check for compliance. This checking may involve applied research into areas of computer science. This research produces new computer science results.

Characterisation 17 *The resident computation scientist builds new computation models and theories.*

Characterisation 18 *Theory checking builds on the laws of mathematics.*

2.3 Programs and Software

We have taken for granted definitions of 'programs' and 'software'. We now seemingly repair that omission.

Definition 7 *By a computer 'program' is meant a set of 'instructions' capable, when incorporated in a 'machine-readable' form, of causing a 'machine' having 'information-processing' capabilities to indicate, perform or achieve a particular function, task or result.*

A program is a single, indivisible structure which prescribes an algorithm over some data structure. When speaking of a program we speak of its "internal" qualities.

You observe that we have defined one concept, program, in terms of at least four other (undefined) terms! For the time being we rely on your previous knowledge of what a program is — and otherwise beg your indulgence.

Definition 8 *By 'program description' is meant a 'complete' 'procedural' presentation in verbal, schematic or other form, in sufficient detail to determine a set of instructions constituting a corresponding computer program.*

Definition 9 *By 'supporting material' is meant any material, other than a computer program or a program description, created for aiding the understanding or application of a computer program, for example problem descriptions and user instructions.*

Definition 10 *By computer 'software' is meant any or several of the items referred to in the previous three definitions.*

A software, or programming system is a collection of programs for the specific purpose of solving an externally stated problem. When speaking of software we speak of its "external" qualities.

The above four definitions (with their appended qualifications) were brought so that you understand the difference between program and software, program system and software system.

2.4 The Four Axes

So the four axes of software development (and its professions) are: management (managers), software engineering (software engineers), programming (programmers), and theory checking (computation scientists).

Oftentimes one and the same person plays all four rôles, sometimes unaware, sometimes frustratingly conscientious of the overlap.

3 Software Quality Measures

SO WHY ARE WE DOING ALL THIS SCIENCE "STUFF"? WHY DO WE HAVE ALL THESE THEORIES, AND WHAT'S THE GOOD OF IT?

We do it, and have it, for three reasons: (i) because we want to understand what we are doing, (ii) because we want to make sure that we are doing it right, and (iii) because it is fun knowing and doing it right!

But what do we mean when we say 'right'; how do we "measure" that which is right, and how much right it is? We do it by subdividing what has to be "measured" into separate, hopefully quantifiable "issues", or qualities. We divide these first into product and project qualities, then each of these fields into sub-fields, etc.

3.1 Software Product Qualities

Here we are interested in describing qualities of software products. We list 8 quality criteria. Except for the last, we claim them independent of one another. Thus we can speak of for example 'correctness' independent of 'reliability', etc.

1: Fitness for Purpose :

> **Definition 11** *Software is fit for its purpose if it meets the following kinds of more or less independent expectancies:*
>
> - *each concept of a "real (or perceived) world" is mapped one-to-one into functions and facilities of the software,*
> - *the software addresses ('solves', computes) the customers problem,*
> - *it is reasonably easy to train the users of the software,*
> - *the software can be used commensurately easily (operativeness),*
> - *the software is ergonomically adequate (physically user friendly), and*
> - *the software is adequately concise, where relevant, in its observable behaviour.*
>
> Some people prefer to use the overall, encompassing characterisation: *user friendly human computer or man-machine interface (HCI, resp. MMI)* for a subset of the above facets. We prefer to be precise so that we can "measure". We also find that the first facet "drives" most remaining: isomorphism between external world concepts and software functions and facilities favourably determines most remaining facets.

2: Correctness :

> **Definition 12** *Software is correct if it meets its functional specification, ie. if and only if the realisation can be proven correct wrt. the specification.*

3: Reliability :

> **Definition 13** *Software is reliable if it clearly prescribes the rejection of invalid input data.*
>
> Typically a compiler rejects syntactically incorrect programs.
>
> A functional specification defines behaviour of a system wrt. acceptable input, but just specify **undefined** or **error** for unacceptable input. A reliable system would safeguard the system against such undefined, erroneous input — while a fault tolerant system, see next, would go further.

4: Fault Tolerance :

Definition 14 *Software is fault tolerant if it clearly prescribes "repairs" to erroneous input, and/or to spurious changes in internal data values.*

Typically a compilers' error-correcting parser 'recovers' from 'minor' errors in input programs.

5: Security/Integrity :

Definition 15 *A software system is secure if an un-authorised user, while anyway using the system (i) is not able to ascertain* **what** *the system is doing, (ii) is not able to find out* **how** *it is doing it, (iii) is not able to prevent the system from operating, and (iv) does not know whether he knows!*

Software integrity must be designed directly into the product, already from its identification, and certainly in its function specification.

Software is maintainable if it is extensible and repairable.

Software extension involves "adding" new functionalities to the software.

Repairability involves three rather distinct issues: software may need **perfective** maintenance in order to improve its performance, **adaptive** maintenance in order to fit it to a changing environment, and **corrective** maintenance in order to repair 'bugs'. In all instances 'maintenance' implies changing the software.

When maintenance of a building for example requires finding the exact location of pipes embedded in walls before drilling into these, then engineers' blue prints are inspected before drilling, and when walls are planned to be torn down, then engineering calculations are (re)done to see whether crucial support goes below set standards.

So it is with software: its maintenance may require inspection of any number of development steps, from requirements, via specifications and abstract designs to concrete designs to locate points of 'maintenance'.

Somehow that for which maintenance is enacted defines a measure of complexity against which the effort (time, cost, etc.) of carrying through the maintenance can be compared. Hence:

6: Maintainability :

Definition 16 *Software is said to be maintainable if the complexity of the 'thing' for which maintenance is enacted stands in some reasonable relationship to the effort of carrying through the maintenance.*

Operationally speaking we can say that if the cost of maintenance is directly proportional to that (otherwise undefined) measure of complexity, then the software is maintainable.

Assume a succession of changes: from the first point in the chain of development documents — where we see that a maintenance change need be expressed (requirements definition - function specification - abstract design - $\cdots$ - concrete design) —

to where the change eventually finds its change in code. If this propagation can be simply ("linearly") contained (and does not, for example "mushroom" all over the code), then the software is maintainable. The preceding is, unfortunately, not a very good characterisation. A better definition would require a longer discussion.

7: Portability :

Definition 17 *Software is portable if it can be "moved easily" from one computing system to another.*

The above "easiness" has not been defined. Hence the definition is meaningless. The point is this: when we define such things as maintainability and portability, then we first need erect a whole set of auxiliary, and properly defined notions — as was implied in the definition and characterisation of maintainability (viz.: complexity, homomorphic, linear, measure, propagation, etc.). For portability we need a similar set of auxiliary concepts — basically amounting to the same as for maintainability.

Porting software is a special kind of adaptive maintenance of software.

A compiler is portable if it can be moved from one operating or machine system to another — and this kind of portability is called re-hosting. If the compiler can be easily changed so as to generate code for another than the originally intended target machine, then the compiler is said to be re-targetable. So compiler portability implies the sum of two things: re-hostability and re-targetability.

8: Robustness :

Definition 18 *Software is robust if its maintenance does not impair any of the qualities mentioned in this section — incl. robustness!*

3.2 Software Development Project Qualities

Here we are interested in qualities of the process by which software is developed.

1. Planning :

Definition 19 *A project can be planned if a method can a-priori be identified, a method which in deterministic steps of development concisely prescribes how to develop the product.*

2. Estimation :

Definition 20 *A projects' consumption of resources (time, manpower, machine and other development tools) can be estimated if for each sub-task that the method (any method) requires, the effort to carry through this task can a-priori be established.*

3. **Resourcing** :

   Definition 21 *A project is resourcable if expected and available resources can be "nicely" mapped onto required resources.*

   The crucial term here is 'nice'. We did'nt define it! If the resources required during a project does not vary dramatically, but can be built up, kept steady, and subsequently tapers off, in an orderly fashion, then the project is resourcable. Resourcability thus depends on the nature of the artifact being built — but knowledge about resourcability can be ascertained once initial planning and estimation has taken place, ie. early!

4. **Economic** :

   Definition 22 *A project is economic, firstly, if the estimated required resources are within expectations, and, secondly, if the actual resources consumed (ie. accounted) stays within budget!*

5. **Trustworthy** :

   Definition 23 *A project is trustworthy if (i) the development staff and the customer at all times believes that management is in full control, (ii) if the developers believe that their method will bring them through the project, and (iii) if the customer believes he will get the quality software at the time he expects.*

6. **Enjoyable** :

   Definition 24 *A project is enjoyable if the development staff has intellectual fun, and is being further educated in pursuing it.*

4 Methods and Principles

4.1 Methods

Definition 25 *A method is a set of procedures for selecting and applying, according to a number of principles, a number of techniques and tools in order to efficiently achieve the construction of a certain, efficient artifact.*

So a method is an orderly arrangement, a procedure or process for attaining an object; a systematic procedure, technique, or mode of inquiry employed by, or proper to a particular discipline; a systematic plan followed in presenting or constructing material; a body of skills and techniques; and a discipline that deals with principles and techniques.

Definition 26 *Methodology is the study of (knowledge about) methods — not just one, but a family, and not just an individualised knowledge, but also a comparative knowledge.*

So a methodology is a body of methods, rules and postulates employed by a discipline; or the analysis of the principles of inquiry in a particular field.

The JSP (Jackson Systems Programming) is a method for constructing a class of software applicable in the context especially of sequential file processing.

The JSD (Jackson Systems Design) is a method for constructing a class of software applicable in a context which extends JSP to parallel (or concurrent) process oriented transaction processing, including what would otherwise be classified as database processing.

Other, claimed, methods are: VDM, SADT, Yordon, etc. Some are formal, some are ad hoc; some cater for a narrow spectrum of either the software life cycle or the application field, or both.

4.2 Principles

Definition 27 *A principle is a comprehensive and fundamental law, doctrine, or assumption; the laws, or facts of nature or mathematics, underlying the working of an artificial device; or a rule, or code, of conduct.*

Let us, however, already here try to identify some such principles — in the form of itemized lists[3] which we do not presently comment upon:

General Principles :

- Analysis of existing artifacts and theories
- Combination of design and analysis
- Decomposability and reducibility
- Abstraction
- Limits of scope and scale
- Divide and conquer
- Impossibility of full capture
- ...

Principles of Philosophy :

- Prevention is better than cure
- From systematic via rigorous to formal approach
- Prove properties rather than test for satisfaction
- ...

Concrete Principles :

- Iterative nature of development

[3]These lists are drawn, directly from: *Software Engineering Education*, eds.: Norman E. Gibbs and Richard E. Fairley, Springer-Verlag, 1987, pp 370-371.

- Representational and operational abstraction
- Denotational vs. computational semantics
- Applicative vs. imperative function definition
- Hierarchical vs. configurational development and presentation
- Discharge of proof obligations
- ...

Principles of Methodology :

- Reduction principle: the whole = the sum of the parts
- Discrete nature of software development: case analysis, induction, abstraction
- Embedded nature of software: impact of context, environment, and enclosing system
- ...

4.3 Techniques and Tools

The computer and computing sciences work bottom-up: provide simple, sometimes elegant techniques and tools that hopefully fit into some method, and hopefully can be used according to some principles.

The computation sciences appear to have a very long way to go before they seem ready to maturely approach the problem of synthesizing methods and enunciating clear principles.

Meanwhile many ad hoc tools will be developed, and hundreds of thousands of hours will be spent on techniques without principles, and on methods with no theories.

5 Conclusion

We have set the stage for what we believe are relevant issues in software development. We have not solved any problems of technical nature. But we have resolved what it is we should be looking for when methods, principles, techniques, and tools are claimed useful in software development.

They have to help guarantee product and project qualities, and they have to fit into (reflect, or mirror) a world consisting of managers, engineers, programmers and theory checkers.

The note can be construed as advocating more unity and clear direction in the fields of computation sciences in support of engineering.

THE SEMANTICS AND COMPLEXITY OF PARALLEL PROGRAMS FOR VECTOR COMPUTATIONS. Part II

by

E.K. BLUM

Abstract

Recent research in parallel numerical computation has tended to focus on the algorithmic level. Less attention has been given to the programming level where algorithm is matched, to some extent, to computer architecture. This two-part paper presents a three-level approach to parallel programming which distinguishes between mathematical algorithm, program and computer architecture. In part I, we motivate our approach by a case study using the Ada language. In part II, a mathematical concept of parallel algorithm is introduced in terms of partial orders. This serves as the basis of a theory of parallel computation which makes possible a precise semantics and a precise criterion of complexity of parallel programs. It also suggests some notation for specifying parallel numerical algorithms. To illustrate the ideas presented in part II, we concentrate in part I on parallel numerical computations which have vector spaces as their central data type and which are intended to be excuted on a multi-processor system. The Ada language, with its task constructs, allows one to program computer algorithms to be executed on multi-processor systems, rather than on "vector (pipelined) architectures". To provide a concrete example of the general problem of programming parallel numerical algorithms for multi-processor computers, part I includes a case study of how Ada can be used to program the solution of a system of linear equations on such computers. The case study includes an analysis of complexity which addresses the cost of data moment and process control/synchronization as well as the usual arithmetic complexity.

Part I appears in an issue of BIT dedicated to Peter Naur on his 60th birthday. It is my pleasure to dedicate part II to Jaco de Bakker on the occasion of his 25th anniversary at the Mathematisch Centrum.

1. Introduction

Many computational systems involve operations on vector quantities which can be executed in a concurrent mode. We consider the specification/design of such systems. In our view, specification takes place on three levels : (1) the mathematical algorithm level; (2)the program level; and (3) the computer architecture level. Actually, this view is an idealization of the practical cases in which it is not always possible to separate the three levels of specification, either logically or in order of performance. Nevertheless, we shall treat them as distinct levels.

We assume that specification starts on level 1 with some mathematical equations expressing the system input-output function. For example, the solution of a system of linear equations is specified by the equations $Ax = b$ and $x = A^{-1}b$, where A is a nonsingular $n \times n$ matrix and b is an n-dimensional vector. The equations involve basic operations on basic sets in one or more data types; e.g. the operations in numerical computations are the arithmetic operations on integers and reals and the boolean operations on {0, 1}. However, as in our example, they may include operations on vectors and matrices. Such a purely mathematical specification may be regarded as taking place on the highest level, which we shall call level 1a. On level 1b, the mathematical equations are transformed into equations which specify a mathematical algorithm. In our example, these are the equations defining the familiar Gauss elimination algorithm. (See part I.) The notion of parallel mathematical algorithm needs to be made more precise and we do this in section 2 in a fairly general context. In numerical algorithms involving vectors, a natural parallelism is often implicit in the operations on components. However, there are usually several different modes of parallel execution possible. To achieve the most efficient usage of the architecture available on level 3, it may be necessary to specify explicitly which parallel mode is to be selected.As yet, there is no standard notation for this kind of specification.

2. The three-level way

As remarked above, we wish to propound the thesis that programming of numerical algorithms is one level of a three-level specification procedure. In this section, we shall make more precise our general conception of levels 1 and 2. On level 1, a mathematical specification of a problem is formulated using mathematical concepts and notations. Level 1 is usually separable into two sub-levels, 1a and 1b. On level 1a, a non-algorithmic description of the problem is given, generally as a set of equations to be solved. The equations involve formulas built up in conventional informal notation from variables, constants and operators in prescribed data types. For numerical problems, it suffices to limit the discussion to what we shall call the numerical data types. These comprise the "standard" types, integer, real, Boolean and real array. The operations in these types include the "standard" arithmetic operations on integers and reals, the algebraic operations on vectors and matrices of linear algebra and the "standard" Boolean operations. We need not be more precise than this for our purposes. In our example, the equations are written in the familiar form $Ax = b$, where A is a given nxn non-singular matrix and b is a given n-vector. The solution can be expressed by the equation $x = A^{-1}b$. It is well-known that this equation does not lead to an efficient algorithm for computing x. If we postulate that A can be factored so that $A = LU$, where L is lower triangular and U is upper triangular, then the first equation becomes $LUx = b$. Setting $Ux = y$, we obtain the equivalent pair of equations $Ly = b$, $Ux = y$, which, with $A = LU$, define the problem in a way that leads to an efficient algorithm. As we know, it is generally necessary to apply a permutation matrix, P, to A before it can be factored. So actually, $PA = LU$ and pivoting must be done in the algorithm. This is the kind of mathematical formulation that should be done on level 1a prior to designing a numerical algorithm.

On level 1b, a mathematical algorithm (m.a.) is designed to compute the solution specified by the level 1a equations. The m.a. is likewise expressed in conventional notation using equations, except that the equations must explicitly define their left-hand sides and, for vector problems, logical quantifiers are used to specify iterations. In this example, the mathematical algorithm is the familiar Gauss elimination method for computing the LU factorization and then solving the triangular algorithms also require Boolean conditions expressed by if-then clauses and Boolean or, and, not operations. The equations and conditions again involve formulas constructed from variables and operators in the data types. Although the variables are to be associated with values in the data types, there is no connotation of storage of values. In particular, there is no notion of "old" value and "new" value of a variable, x. These are level 2 concepts. On level 1. to denote the changing values caused by a recursive iteration, an index is associated with x; e.g. x(i-1) represents the old value of x at the beginning van the ith iteration and x(i) the new value after this iteration. There is no dynamic operation like an assignment, which changes the "state" of a "memory location" for x. Memory is a level 2 concept. Thus, level 1b is essentially applicative (i.e. functional). Although there are formal applicative languages like the lambda calculus and much ongoing research into functional programming, at present we usually rely on traditional mathematical notation enriched with special notation to specify m.a.'s of the kind encountered in vector computations. Since there are no standard notations for specifying the parallel aspects of such m.a.'s, we present some of our own in part I.

What we have called "traditional" or "conventional" mathematical notation has evolved over the last three centuries, with many conventions already present in the works of Descartes, Pascal and Fermat. This notation, with its formulas involving numerical constants, letters for variables, symbols such as "+" for operations, superscripts for exponents and punctuation such a parentheses has been absorbed into most modern programming languages. Given a precise syntax (e.g. in Backus-Naur form) in languages like Fortran, Algol, Pascal and Ada, formulas have a

structure representable by parse-trees. The parse-tree embodies a partial ordering of the "applications" of operations to operands.

The parse tree also specifies a partial order subexpressions which corresponds to an ordering of the functional combinators which perform composition and tupling of functions. The functional structure implicit in a parse-tree defines a "data flow" relation between the results of certain operations and the "input" operands of other operations. Indeed, this non-algorithmic functional aspect of formulas is their essential mathematical meaning. Traditional notation also includes equations, in particular, those of the form "variable = formula". A set of such equations may specify a partial order of applications which is not a tree.

The functional structure of formulas and equations is "static". Algorithms involve something more, namely, a dynamic aspect that translates into a temporal partial ordering of the applications of operations, i.e. a "control-flow". In the classical formal notions of algorithm (e.g. of Turing, Post, Markov), which are not concerned with efficiency and complexity, the control flow is essentially sequential (i.e. totally ordered). To deal with efficiency, complexity and parallelism we shall reformulate the notion of algorithm using partial orders of applications. Although these partial orders are implicit in the traditional notation for formulas and equations and would seem to be a consequence of the data flow ordering, this becomes less obvious when logical conditions and iteration specifications are included. In fact, when level 3 computational constraints are imposed, such as number of processors, speeds of operations and inter-processor communication, the partial order may not be entirely a consequence of the static functional structure of formulas. To prepare for this level 1b, attention shoud be focused on how an m.a. should be composed out of _partial orderings_ of _applications_ of operations to operands. Thus, level 1b is intermediary between the functional approach on level 1 a and the typical programming approach on level 2.

On level 2, the m.a. is transformed into a _computer algorithm_ described by a computer program. It is generally accepted that

most programming languages provide practical counterparts to the classical formal systems for specifying "effective procedures".

A _computer algorithm_ is an effective procedure constructed from a _basis_ of operations provided by the particular languages used to write the program. Although the basis differs from language to language, there is a common core, some of which we have already mentioned (e.g. the standard Boolean and arithmetic operations). Aside from recent functional programming languages, most languages provide an _assignment_ operation which changes the state of an _abstract store_ (memory) associated with the variables in a program. There is also an implicit _finite-state control_ unit, as in a Turing machine, and many of the basis operations (e.g. goto) are for the purpose of manipulating the _control state_. The transformation of an informal specification of a level 1b mathematical algorithm into a formal computer program specifying a computer algorithm on level 2 is the essential task of the computer programmer.

Most programming languages are not applicative with respect to the operations which manipulate the state of the store and the controls state, since there are no state variables of type state that can be used explicitly in programs. For example, in an application of the assignment operation, the "current" store state is an implicit operand and the "next" store state is an implicit result. A goto is an operation on the implicit control state which has a new control state as result. In the multi-task programs in Ada, the control state can have a complex structure distributed over many concurrent tasks. We note that the synchronization primitives of Ada permit quasi-orders rather than partial orders of applications; i.e. it can happen that deadlocks occur. Presumably, such deadlocked programs are incorrect as far as m.a.'s are concerned. A parallel computer algorithm can be regarded as a system of concurrent "processes". The simplest kind of process is the sequential algorithm. How processes are to be joined together into a system of concurrent processes which interact is still a subject of resarch. In the three-level way, a programmer faced with the problem of transforming a given m.a.

(specified in some informal notation) would analyze the m.a. into purely sequential m.a.'s combined using cominators for composition, paralleling and recursion. These combinators are usually realized as synchronization and control operations on level 2. We illustrate this in our Ada case study.

Finally, on level 3, the computer algorithm is transformed into a specific computer system. A computer system is characterized by its architecture which configures a set of hardware and software components.

Some of the more routine difficulties in programming are being ameliorated by "tools" that increase the skill of the programmer in dealing with complex syntactic details and in manipulating files. However, in complex problems, the major difficulties lie elsewhere and are primarily semantic. From the perspective of the three-level way, we identify the following sources of semantic difficulties in the programming of parallel algorithms, which may occur in the sequential case as well.

1) The mathematical algorithm that a programmer starts with may nog be precisely specified and conventional mathematical notation may be inadequate to the task.

2) The mathematical algorithm may be incompletely specified, in which case the specification must be completed on level 2 as part of the programming.

3) The constructs provided in the programming language may not match the mathematical algorithm constructs, making the level 1b-to-level 2 transformation a complex one.

4) Programs describe computer algorithms to be executed on some real computer system, hence must include computer-oriented specifications (e.g. date storage) that permit them to be translated into a level 3 specification that can be executed. The inclusion of these details is often tantamount to the design of a virtual computer system.

5) The level 2-to-level 3 translation is performed by a compiler and a run-time system that must deal with bounded resources (memory, processors, timing constraints, communication bandwidths, interconnections, etc.) and the problems raised by the general-purpose stored-program concept.

To achive efficient compilation the programmer must often provide further implementation-dependent information on level 2. This many have to be done in an environment that is an extension of the formal framework of the programming language, requiring the programmer to attend to level 3 matters as well.

6) For parallel algorithms there is the further difficulty that there is no generally accepted formal model analogous to those for sequential algorithms.

Ideally, the solution to these difficulties is to eliminate level 2 entirely. The historic trend to higher-level languages, which made great early strides when assembly languages were replaced by Fortran, Algol, Pascal etc. has failed to come even moderately close to this ideal with more recent languages. In fact, for parallel algorithms, the quest for maximum speed and efficiency may make this ideal unattainable. However, it seems possible to greatly reduce the effort on level 2 by attacking the deficiencies on all three levels.

3. Mathematical algorithms

If f is an n-ary operation and $a_1,\ldots,a_n$ are valid operands in its domain, then an application of f to these operands yields a result, $f(a_1,\ldots,a_n)$, in one of the numerical data types. For brevity, we shall use vector operands, writing a for $(a_1,\ldots,a_n)$ and f(a) for the result. We shall also allow vector variables as operands. As a simple example, 2+3 denotes the application of the integer add operation to the operand (2,3) with the result 5. (Usually, we use infix notation.). 3+2 is a different application of + with operand (3,2) and the same result. An expression containing variables will be regarded as (denoting) a family of applications. We call such a family a symbolic application. Thus, x+3 denotes a symbolic application which is the family of all applications consisting of adding 3 to an integer. In general, a symbolic application, r(x), parametrized by $x = (x_1,\ldots,x_n)$ is to be regarded abstractly as a mapping $r\colon D_1\times\ldots\times D_n \to Ap$, where x_i varies over data type D_i and Ap is the set of all applications of some operation, ω_r. For any value $v = (v_1,\ldots,v_n)$ with $v_i \in D_i$, the mapping yields an application, r(v), in the family. r(v) consists of operation ω_r, an input operand, In(r(v)), and a result, Res(r(v)). If In(r(v)) is in the domain of ω_r, then we require that $Res(r(v)) = \omega_r(In(r(v)))$ and otherwise $Res(r(v)) = \perp$, where $\perp$ is a special element read as "undefined". Thus, if r(x) is given by the formula x+3, where + is integer addition, then r(2) is the application with In(r(2)) = (2, 3) and Res(r(2)) = 5. So r(2) = (+, (2, 3), 5). On the other hand, $In(r(\sqrt{2}))$ is $(\sqrt{2}, 3)$ and $Res(r(\sqrt{2})) = \perp$. In most of this paper, symbolic applications will be specified by conventional mathematical formulas and equations, but this does not preclude their representation by other means, such as tables, for example. For convenient reference, we summarize the foregoing in a definition.

Definition 1. An application, α, is a triple consisting of an operation ω_α, an input, $In(\alpha)$, and a result, $Res(\alpha)$, such that $Res(\alpha) = \omega_\alpha(In(\alpha))$ whenever $In(\alpha) \in Dom(\omega_\alpha)$ and otherwise $Res(\alpha) = \bot$. A symbolic application is a family of applications all having the same operation.

A symbolic application represents a function's graph, but note that the functions involved may be different even when the same operation is used, as in x+2, x+3 and x+y. In algorithms, applications occur in a dynamic context that must be specified. For example, the formula (2+3)*4 can be interpreted a a dynamic sequence of two applications, 2+3 followed by 5*4. The formula, (3+2)*(2+4) denotes either of the sequences (3+2, 2+4, 5*6) or (2+4, 3+2, 5*6 to compute the result, 30. This ambiguity can be used to advantage in paralle algorithm specification. On level 1a, we interpret a mathematical formula as denoting a function and a partial order of functional (data-flow) dependencies. On level 1b, we shall interpret such a formula as suggesting a partial order (p.o.), of applications in a temporal sense. In simple cases, this p.o. can be represented as a finite tree. In the preceding example of formula (3+2)*(2+4), we have the familiar tree,

Figure 1

(Note the implied order of the operands in such a figure.)

Suppose p and q are two p.o.'s on a set S. We say that p is a refinement of q if $q \subset p$ as binary relations. Let φ be a conventional formula and $\leq_\varphi$ the p.o of its parse-tree. If p.o.'s p and q are refinements of $\leq_\varphi$, then executing the applications according to p yields the same results as

executing them according to q. For example, both of the application sequences given above are refinements of the p.o. in Figure 1 and yield the same result. We shall use $\leq$ to denote partial orders. We say that an application, r, is a <u>predecessor</u> of application s in a p.o. if $r \leq s$. It is an <u>immediate predecessor</u> if there is no u such that $r \leq u \leq s$.

A formula that contains a (vector) variable $x = (x_1, \ldots, x_n)$ can be interpreted as denoting a set, F(x), of symbolic applications, each parametrized by zero or more of the x_i variables, and having a partial order specified by the structure of the formula. For example, let f(x) be the formula (3+x)*(x+4). The p.o. which f(x) denotes can be represented by a tree having the structure,

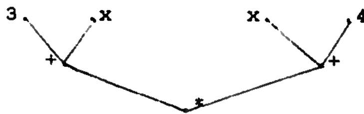


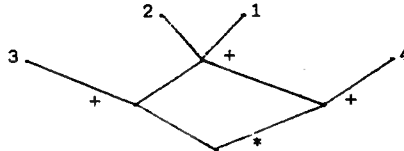
Figure 2

where the parameter x can have any integer value. The set, F(x), is the set {x, 3, 4, 3+x, x+4, (3+x)*(x+4)}. (We shall consider constants and variables to be 0-ary operations.) Note that the application (3+x)*(x+4) is obtained by tupling of the applications 3+x and x+4 followed by composition with the application y*z. Later, we shall replace tupling by a parallelling combinator and composition by a sequencing combinator. The tree represents a partial order, $\leq$, on the symbolic applications in F(x). If v is an element in a data type, then F(v) is the tree obtained by substituting v for x. There is a partial order, $\leq_v$, on F(v) induced by $\leq$ in a natural way; e.g. F(2) and $\leq_2$ are given by Figure 1 in this example.

Partial orders have been considered in a variety of contexts as a basis for the semantics of concurrent processes. What is new in what follows is the choice of algebraic applications, suitably indexed, as the elements of a

partial order in a natural way that leads to a precise definition of mathematical algorithm and algorithmic complexity. This also seems to provide a natural mathematical framework for both the static and dynamic structure of parallel algorithms and their computations. We treat applications and partial orders abstractly, rather than syntactically, to allow for a variety of notations on level 1b, including some which may not fit into the usual algebraic syntax of terms. Furthermore, as we wish to give a dynamic interpretation of applications occurring in a temporal partial order in a computation which may be specified by expressions combined with equations and various logical and control operators, we cannot restrict the orders to trees. For example, although the pair of equations, x = 2+1, y = (3+x)*(x+4) can be combined into one with a single right-side term (tree), they can also be interpreted to specify the following algorithm: compute x as the result of the application 2+1, substitute the result, 3, for x in the formula for y, then do the set of applications {3+3, 3+4} in some order (possibly in parallel) and finally do 6*7. This p.o. of applications is not a tree. It can be depicted by a kind of Hasse diagram as in Figure 3. (Downward paths specify the p.o..)

Figure 3



Observe that this p.o. has several <u>minimal</u> elements (1,2,3,4) to start the computation. Compare figure 3 with Figure 4, which is the tree obtained from Figure 2 by tree substitution of 2+1 for x.

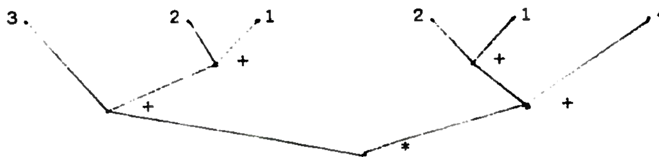


Figure 4

The algorithm depicted in Figure 4 computes the result 42 also, but has two instances of the application 2+1. In transforming this algorithm into a parallel computation, we shall interpret this to mean two executions of the same application. In this case, this is clearly avoidable a priori. However, it is possible that at level 2 such duplication is actually more efficient when executed on two processors if data access operations are included in the complexity analysis. (They can also be included on level 1b by using the identity operation. See below.) In general, such repetition is unavoidable a priori. For example, in the formula (3+(2+x))*((2+y)+4), it may happen that x and y are both set equal to 1 in other equations. To distinguish instances of the same application, we are led to consider indexed sets of applications. The indexing could be done with reference to the syntactic structure of the formulas and equations which specify an algorithm, but, again, we do not wish to restrict the method of indexing to be constrained by conventional syntax as there may be other ways to specify the "control flow". Hence, we shall allow arbitrary posets as indexing sets.

Definition 2. Let $(J, \le)$ be a poset (the indexing poset) and F(x) a set of symbolic applications, each parametrized by zero or more variables in the sequence of distinct variables, $x = (x_1, x_2, \ldots)$. An algorithmic structure (a.s.) on F(x) is given by an indexing function, ap: J -> F(x), which defines an indexed poset, $(F(x))_J$, of symbolic applications. We denote the a.s. by $((F(x))_J, \le)$. Further, for each value sequence, v, and $r(x) \in F(x)$, let r(v) be the corresponding application. Define F(v) to be the set of all r(v) such that $Res(r(v)) \ne \bot$. Let $Jv \subset J$ consist of the indices, j, for which $ap(k)(v) \in F(v)$ for all $k \le j$. Define an indexed poset, $(F(v))_{Jv}$, by the derived indexing function ap_v: Jv -> F(v) given by $ap_v(j) = r(v)$, where $r(x) = ap(j)$. Let $\le_v$ be the p.o. which is the restriction of $\le$ to Jv. We call $((F(v)_{Jv}, \le_v)$ a (parallel) computation structure (pcs) of the a.s. .

Remark. It is convenient to say that application ap(j) precedes ap(k) when $j \le k$. The variables $x = (x_1, x_2, \ldots)$ which parametrize the symbolic applications in F(x) take values in data types D_i. Each symbolic application, r(x), is a family defined by a mapping, $r: D_{i_1} \times \ldots \times D_{i_n} \to Ap.$, on finitely many of the data types. We say that r(x) <u>depends on</u> $x_{i_1}, \ldots, x_{i_n}$. It is convenient to regard each variable, x_i, as a symbolic application which is a family of 0-ary applications. For a sequence of values ,v, the application $x_i(v)$ is the 0-ary operation v_i. In most algorithmic structures, the parametrizing variables x_i will be included in the structure as 0-ary applications which are predecessors of certain applications. This will be the case when ordinary formulas are used to specify the structure. We call such x_i <u>input variables</u>. In this paper, we shall assume that all x_i which parametrize an a.s. are input variables. For each (input) value, v, the results of certain maximal applications (if any) can be indexed as <u>outputs</u>. To output an intermediate result (of a non-maximal application), we can force it to be the result of a maximal application simply by adjoining the identity operation.at that point.

An a.s. specifies permissible "control flow" or execution dynamics of a computation. It is also necessary to specify functional structure ("data flow") as would be implicit in a parse-tree. In an a.s., there must also be some relation between the inputs and results of its various applications whereby the needed inputs of any application are the results of applications which are predecessors. This is provided in the next definition.

<u>Definition</u> 3. An algorithm structure $((F(x))_J, \le_J)$ is said to be a <u>mathematical algorithm</u> (m.a.) if it has the following algorithmic properties:

(1) The set of minimal applications is finite;

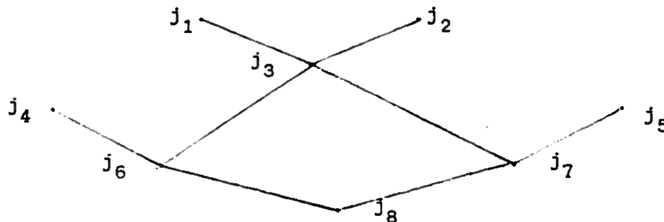
(2) Each application has a finite number of predecesors and immmediate successors;

(3) For each value sequence, v, there exists a data flow function, D_v, defined on J as follows. For $j \in J$, let r(x) = ap(j) have an operation part of arity $n \geq 1$. Then $D_v(j) = (j_1, \ldots, j_n)$, where the j_i are predecessors of j such that $In(r(v)) = (Res(ap_v(j_1)), \ldots, Res(ap_v(j_k)))$.

If $\leq_J$ is total, the m.a. is called sequential and if p is a total order which is a refinement of $\leq_J$, then the m.a. is said to be sequentialized by p.

To illustrate these concepts, let us define an a.s. for the formula (3+(x+y))*((x+y)+4). Let J be the poset given by the diagram in Figure 3a.

Figure 3a



An a.s. is defined by the indexing $ap(j_1) = x$, $ap(j_2) = y$, $ap(j_3) = x+y$, $ap(j_4) = 3$, $ap(j_5) = 4$, $ap(j_6) = 3+(x+y)$, $ap(j_7) = (x+y)+4$, $ap(j_8) = (3+(x+y))*((x+y)+4)$. For v = (2,1), we get the poset F(v) in Figure 3, which is isomorphic to Figure 3a since all symbolic applications are defined for v = (2,1). The data flow function D_v is defined for each j in J as the pair of immediate predecessors when ap(j) has a binary operation.

It follows from algorithmic property 3 that every minimal application is

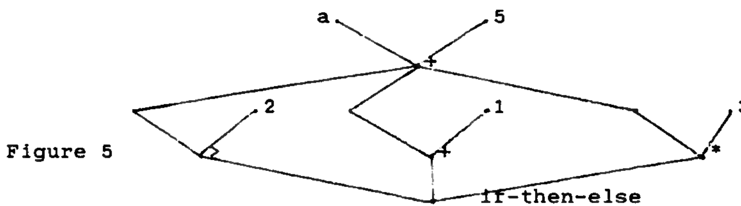
0-ary, since otherwise it would have at least $n \geq 1$ predecessors. From property 1 it follows that there are a finite number of starting points for a computation of an m.a. .From property 2 it follows that at any point in a computation there are only a finite number of parallel paths. We see that an m.a., $((F(x))_J, \leq_J)$, gives rise to a parametrized (by the values of x) family of indexed posets of applications. If we think of each indexed application as residing in a micoprocessor able to execute its operation, each such indexed poset, $((F(v))_{Jv}, \leq_v)$, suggests various possible dynamics (temporal orders) of execution of the applications in it. If $j \leq_v k$, then $ap_v(j)$ should not (usually cannot) be executed later than $ap_v(k)$. This insures that all results needed as inputs to an application are computed prior to its execution. (The microprocessor for $ap_v(j)$ would have to be connected to the one for $ap_v(k)$ to allow the proper data flow.) An execution of all the applications $ap_v(j)$, $j \in Jv$, satisfying these temporal constraints is a <u>parallel computation</u> of the m.a. for input data v. The p.o. $\leq_J$ allows different dynamical modes of parallel execution. These modes correspond to different refinements of $\leq_J$.

For example, consider the equations $x = a+1$, $y = (3+x)*(x+b)$. An m.a. which they specify can be represented by a diagram obtained from Figure 3 by replacing the nodes 2 and 4 by a and b respectively. a and b are input variables. Assigning the values 2 and 4 to a and b respectively yields the pcs in Figure 3, which defines the possible parallel computations of the algorithm for these input data. This is a simple example, of course. In the case study,we shall consider a real algorithm in which vector and recursive iterations occur.

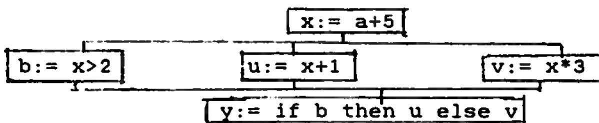
As yet, we have said nothing about the effective calculability of the indexing function ap and the data flow function D. Indeed, the symbolic application has not been restricted to be effectively calculable (i.e. computable). We defer such issues to the choice of a particular formal notation for m.a.'s.

The branching produced by execution of Boolean conditions is easily accommodated into the partial order structure of an m.a., but, of course, it should not be confused with the forking produced by parallel execution. One way to include (deterministic) branching in an m.a. is to use the if-then-else operation. _If_ b _then_ f _else_ g can be interpreted applicatively as an operation having three operands, b, f, and g. Thus, the two equations, x = a+5, y = _if_ x > 2 _then_ x+1 _else_ x*3, would specify the m.a. depicted below

Figure 5

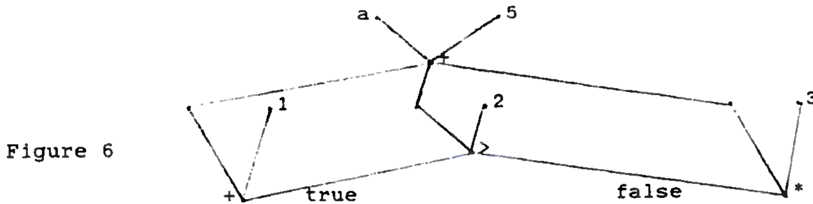


in which if-then-else has the result x+1 if x > 2 and x*3 otherwise, where all x's have been replaced by a+5 (as indicated by the unlabeled nodes). The operations >, +, and * can be applied in any order prior to if-then-else. This parallel interpretation of if-then-else would lead to a level 2 parallel flowchart like



in which there is a three-way fork and join. A more conventional approach to logical branching is simply to consider a symbolic application like x > 2 as denoting two disjoint families of applications, those (like 3 > 2) having the result True and those (like 1 > 2) having the result False. If we denote these families by $r_T(x)$ and $r_F(x)$, then $r_T(3)$ = ((3,2), >, True), $r_F(1)$ = ((1,2), >, False), $r_F(3)$ = ((3,2), >, ⊥) and $r_T(1)$ = ((1,2),>, ⊥). The diagram in Figure 5 would be replaced by the one in Figure 6 below.

Figure 6



The arcs labelled "true" and "false" indicate which application results are to have x+1 as successor and which are to have x*3. Thus, $r_T(x) \leq_J x+1$ and $r_F(x) \leq_J x*3$. According to Definition 3, this p.o. defines an m.a. which has the execution dynamics of the conventional sequential flowchart with a test box for $x > 2$ having a two-branch exit. Finally, in passing, we note that non-deterministic branching in an m.a. can be done by an application of a "choice" function. For example, Choice(x+1, x+3) would allow either x+1 or x+3 as a result, indicating that either application can be performed at the point of the Choice application. Such non-deterministic algorithms can be defined as sets of m.a.'s. Thus, a formula like Choice(x+1, x+3)*(4+x) would specify a set of two m.a.'s, one given by the formula (x+1)*(4+x) and the other by the formula (x+3)*(4+x) as in Figure 2.

By using the branching combinators (both deterministic ones like if-then-else and non-deterministic ones like Choice) together with other combinators, m.a.'s of a complex structure can be composed out of simple m.a's. It seems possible that the static structure of most m.a.'s in practice can be analyzed and synthesized with relatively few combinators, making possible an algebra of m.a.'s analogous to the algebra of processes in [37]. The following combinators seem most natural: *sequencing* (or *composition*), *paralleling* (or *union*) and *recursion* (or *iterated composition*). Various versions of these combinators have been studied in the context of programming languages, flowcharts and processes [Part I: 37, 38, 46-48]. We sketch their definition for m.a.'s. (A complete development is the subject of ongoing

research. We illustrate their use in the case study.)

In terms of ordering, sequencing is a matter of connecting designated maximal elements (the outputs) of a p.o. to designated minimal elements (the input variables) of another p.o. This is easy to visualize when the Hasse diagrams of the p.o.'s are simple. Intuitively, for two m.a.'s, f and g, we can form a new m.a., $f \cdot g$, in which designated output nodes of f are connected to designated input nodes of g. For example, suppose we use a conventional equational specification of an m.a. in which f is given by the equation, $x = 2+1$ and g by $y = (3+x)*(x+4)$. One possible sequencing is given by Figure 3a in which the two instances of the input variable x are replaced by the output node (+) representing the application 2+1. This could be interpreted to mean that the result of 2+1 can be accessed concurrently by 3+x and x+4. To take into account actual data access operations on x (at levels 2 and 3) in which data accesses to x probably occur serially, we may wish to specify an m.a. in which the result of 2+1 (stored in x) is accessed first by x+4 and then by 3+x. To do this, we introduce the identity operation, id , and the application id(x), which has input x and result x. We then form an alternative sequencing, $f \cdot_{\sigma} g$ shown in Figure 3b.

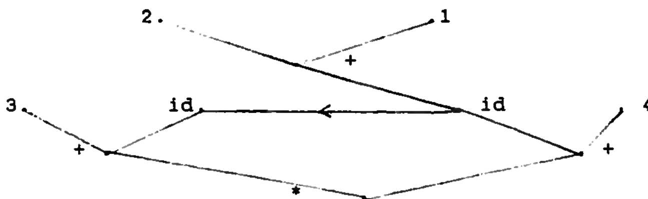


Figure 3b

σ denotes a "connection" mapping which maps the two input (x) nodes of g onto the output (+) node of f. Then the application x is replaced by id. The applications 3+x and x+4 are changed accordingly. Now to specify the serial data accesses, we add a directed edge from one id node to the other, which yields a refinement of the p.o in Figure 3a.

We make these intuitive ideas of sequencing more precise in the following

definition.

Definition 4. Let $f = ((F(x))_J, \leq_J)$ and $g = ((G(y))_I, \leq_I)$ be a.s.'s., with respective indexing functions ap_J and ap_I. We assume $I \cap J = 0$. Let $(r)_{J'}$ be a subfamily of outputs of f and $(y')_I$ a subfamily of input variables of g. Let $\sigma: (y')_I \rightarrow (r)_J$ be a connection function which maps the set $I' \subset I$ of indices of $(y')_I$ into the set of indices $J' \subset J$ of $(r)_J$. Define $G(y|r))_{I|J'}$ to be the indexed family obtained by modifying $(G(y))_I$ as follows: Replace each minimal input variable $ap_I(i) \in (y')_I$ by $ap_J(\sigma(i))$ and replace $i \in I$ by $\sigma(i)$. If $ap_I(i) \in (y')$ is not minimal, replace it by the application which has the identity as its operator and $Res(ap_J(\sigma(i))$ as its input. This yields a new indexing set, $I|J'$, with a p.o. $\leq_{I|J'}$ and a new a.s. $(G(y|r))_{I|J'}$. Let $\leq_{J \cdot I}$ be the p.o. which is the transitive closure of $\leq_J \cup \leq_{I|J'}$. Then the indexed family $(F(x))_J \cup (G(y|r))_{I|J'}$ with $(J \cup I|J', \leq_{J \cdot I})$ as indexing p.o. is an a.s., $f \cdot g$, called a sequencing of f and g.

In a conventional equational specification of an m.a., the variables on the left sides usually prescribe how a sequencing is to be done. Thus, Figure 3 depicts a sequencing of the elementary algorithm 2+1 with the m.a. shown in Figure 2, as specified by the equations $y = (3+x)*(x+4)$, $x = 2+1$. (Our sequencing combinator for algorithmic structures is somewhat analogous to sequencing of processes in [37], where the variable x is said to be "internalized" and output-input matching is modelled abstractly by monoid morphisms and a "restriction" operation. Processes are level 2 or 3 constructs in our approach.) Also note that definition 4 is independent of the choice of a syntax for m.a. specification. This makes it simple to establish the following basic property of sequencing.

Lemma 1. If f and g are m.a.'s, then so is any sequencing $f \cdot g$.

Proof. We simply verify the algorithmic properties (1), (2) and (3) in definition 3. Properties (1) and (2) are immediate from the definitions. To establish (3), we simply define a data-flow function, D_{fg}, for $f \cdot g$ in a natural way. Let v be a sequence of input values. For an application, ap, in f we set $D_{fgv}(ap) = D_{fv}(ap)$, where D_{fv} is the data-flow function of f with the input values, v, replacing the input variables of f. For an application, ap, in g, there are two cases. If all inputs of ap are connected to outputs of f, then the results, w, of these outputs for input values v are considered as input values for g and we set $D_{fgv}(ap) = D_{gw}(ap)$, where D_{gw} is the corresponding data-flow function of g. If some input variable, y_i, of g is left unconnected, then the value v_i is used as the input value of y_i in combination with the values w.

We now give the definition of the paralleling combinator.

Definition 5. Let f and g be as in definition 4. The paralleling of f and g is the a.s., $f \| g$, having (x, y) as input variables and the outputs of both f and g. Its indexing poset is the union $I \cup J$ with p.o. $\leq_I \cup \leq_J$.

This is the purest kind of parallel combinator, since it does not establish any serial connections between f and g. However, it appears to suffice for the numerical algorithms considered in this paper. Of course, when $f \| g$ is sequenced with another a.s., the outputs of f and g may be commingled.

Lemma 2. If f and g are m.a.'s, then $f \| g$ is an m.a. .

Proof: Immediate from the definitions of m.a. and paralleling.

Definition 6. A recursive iteration on a.s., f, is an an a.s., f^{∞}, obtained by a potentially infinite sequence of sequencings of f with "copies" of itself in which each input of one copy is connected in a uniform way to an output of the preceding copy.

Under appropriate conditions on the sequencing, if f is an m.a., then so

is f^{∞}. We shall not prove such a result here , but illustrate it with a parallel m.a. for the factorial function, n! = n*(n-1)!, starting with 1! = 1. In equation form, we specify parallel n! as follows using some notation explained below.

```
FACTORIAL(n) is
    {y(0) = 1, FACT(0) = 1, x = n};
    for i = 0,...,INFINITY loop
    {y(i+1) = if x>y(i) then y(i)+1 else 0,  FACT(i+1)=y(i)*FACT(i)}endloop
```

We use braces to delimit sets of equations that may be evaluated in parallel, whereas the semi-colon denotes sequencing. The <u>for</u>...<u>loop</u>...<u>endloop</u> notation denotes iterated sequencing of the m.a. delimited by loop and endloop. As in conventional usage, the outputs y(i+1), FACT(i+1) are to replace the inputs y(i), FACT(i) of the next iteration. To represent the p.o. of this m.a., consider the equations in the second pair of braces. These define an m.a., f, with three input variables, x, y, FACT, and a p.o. depicted schematically by the solid lines in Figure 7.1. In Figure 7.2, the solid lines depict a copy of f with inputs replaced by Id nodes, indicating connection to the previous output nodes connected by dashed arcs. Output nodes are also labelled Id.

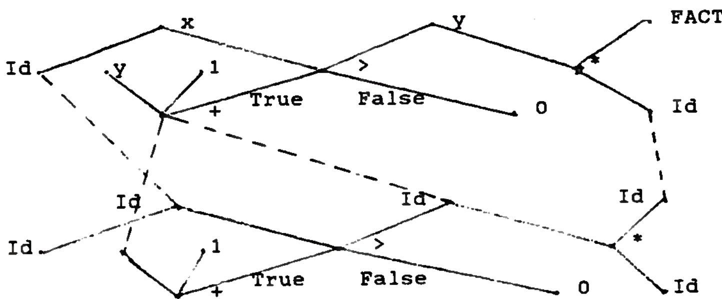


Figure 7.1

Figure 7.2

Together, Figures 7.1, 7.2 depict f·f. An infinite sequence of such connected figures would depict the m.a. f^{∞}, an infinite p.o. which can be defined mathematically as a fixed-point of an appropriate combinator. This is a well-known approach to recursion. (e.g. see [37].) f^{∞} can also be defined set-theoretically, as we have done, as a transitive closure. The construction

of the m.a. for FACTORIAL(n) is completed by the sequencing $f_o \cdot f^{\infty}$, where f_o is the initialization m.a. consisting of the three 0-ary applications n, 1,1 to be connnected to x, y, and FACT in Figure 7.1. Although this is a parallel m.a. for n!, we have used the second interpretation of if-then-else in which x>y denotes two families of applications. Using the value $\perp$ as in Definition 3, each integer value,v, of n determines a finite computation structure, FACTORIAL(v), having v copies of f and the output v!.

We are now able to give a precise definition of the complexity of an algorithm. If an m.a., $((F(x))_J, \leq_J)$, is sequential, it determines a unique computational dynamics for each input value sequence, v, since $\leq_v$ is a total order. The discrete time dynamics can be represented as a sequence of applications (ap(1), ap(2),...), possibly infinite if recursion is involved, where ap(i) is considered to be executed in a time interval $[t_i, t_{i+1})$. If the m.a. is parallel and if $\leq_v$ is not a total order, then any refinement, $\leq_v'$, of $\leq_v$ gives rise to a discrete-time dynamics of execution. We shall define two models of discrete-time dynamics: synchronous and asynchronous. Consider the indexed family $(F(v))_{Jv}$. We partition it into a possibly infinite sequence of disjoint subfamilies S_1, S_2,..., where S_1 is the family of all applications ap(j) such that j is a minimal element of $\leq_v'$, S_2 is the family of ap(j) such that all immediate predecessors of j are (indices of elements) in S_1, S_3 consists of all ap(j) such that j has all its immediate predecessors in $S_1 \cup S_2$ and at least one in S_2 and so on. Thus, S_i contains precisely those ap(j) such that all immediate predecessors of j are in $\cup_{k<i} S_k$ and at least one is in S_{i-1}. Let $t_o < t_1 < \ldots$ be a sequence of time points. In a synchronous dynamics model, all ap(j) in S_i are executed in the interval $[t_{i-1}, t_i)$. Since the ap(j) may not be distinct applications, there can be multiple executions of an application in any interval.

Definition 7. The sequence $S(\leq_v') = (S_1, S_2, \ldots)$ defined above is called the synchronous parallel computation determined by $\leq_v'$ and S_i is its ith step. The synchronous time-complexity of $((F(v))_{Jv}, \leq_v')$ is the length of $S(\leq_v')$. Let R_v be a set of refinements of $\leq_v$. The synchronous time-complexity of the m.a. $((F(x))_J, \leq_J)$ relative to R_v is $C(v) = \min(\text{length } S(\leq_v') : \leq_v' \in R_v)$.

In [49] (Part I), a synchronous model of computation is adopted for measuring complexity. Their model is described in terms of "synchronous processors, each having access to the same storage ... and ... at any parallel step i, any processor may use any input or any element computed by any processor before step i...". Our Definition 7 makes this concept mathematically precise purely in terms of p.o.'s and extends it to include constraints on the p.o.'s. For example, to define the notion of "k-parallelism" [49] we restrict the refinements in R_v to those in which each set S_i has at most k applications. As in most complexity definitions, in numerical algorithms it is usually possible to characterize inputs v according to some size criterion, say size(v), in such a way that complexity is a function of size(v) rather than v itself. For example, in the problem $Ax = b$, we can take the dimension , n, of the vector space as size(v).

In an asynchronous dynamics model, not all applications in S_i are executed in interval $[t_{i-1}, t_i)$. Some may be delayed waiting for operands which are results of preceding "slow" application executions or may themselves be slow. To model this kind of asynchronous dynamic behavior we introduce a duration function, $d: F(v) \to N$, which prescribes an integer time duration $d(r(v)) \geq 1$ for each application. Then the completion time for $ap(j) \in (F(v))_{Jv}$ relative to $\leq_v'$ is the integer $t(ap(j))$ given by

$$t(ap(j)) = \max\{t(ap(i)) : i \leq_v' j\} + d(ap(j)).$$

In synchronous dynamics, $d(r(v)) = 1$ for all $r(v)$.

<u>Definition</u> 8. Let $((F(v))_{Jv}, \leq_v')$ be a parallel computation structure of the m.a. $((F(x))_J, \leq_J)$. Let d_v be a duration function on F(v) and let AS_i be the subfamily of $((F(v))_{Jv}$ consisting of those ap(j) such that t(ap(j)) = i. The sequence (AS_i) is called an <u>asynchronous parallel computation</u> of the m.a.. The <u>length</u> of (AS_i) is the maximum i such that AS_i i nonempty. The <u>asynchronous time-complexity</u>, Ac(v), of the m.a. relative to a set, D, of duration functions and a set, R_v, of refinements of $\leq_v$ is the minimum of the lengths of all asynchronous parallel computations determined b D and R_v.

Most of the theoretical studies of algorithmic complexity are based on the synchronous model of dynamics . Asynchronous complexity studies would probably restrict the set, D, of duration functions. A reasonable restriction would be to functions which prescribe the same duration to all applications having the same operation part or similar operation parts; e.g. all arithmeti operations would have the same duration for all operands. We shall make this kind of restriction in our case study in section 5. We remark, in passing, that if we allow duration functions which are infinite for some applications, then it seems possible to model such properties as safety and liveness of asynchronous computations..This is a matter for future study.

E. K. Blum
Mathematics Department
University of Southern California
Los Angeles 90089, California

March 14, 1989

Acknowledgement: This research was partly supported by N.S.F. grant CCR 8712192

A Summary of the Work on the Proof Theory for the Language POOL

Frank de Boer
Centre for Mathematics and Computer Science
P.O. Box 4079
1009 AB Amsterdam
The Netherlands

March 28, 1989

Under the guidence of Jaco de Bakker I have been working with Pierre America in the ESPRIT project 415 on the proof theory of the language POOL (a Parallel Object-Oriented Language [Am]).

A program of the language POOL describes the behaviour of a system consisting of *objects*. An object operates upon a *local state* which assigns objects to variables. A local state is directly accessible only to the object to which it belongs.

An object comes into existence by some "creative act" of some other object. The identity of the newly created object is stored into the local state of the "creator". When an object is created a local process is started up which then will run in parallel with the local processes of the already existing objects.

Objects interact by some form of *remote procedure call* (also called *rendezvous*). A call consists of a specification by the sender of the receiver, a procedure (also called a *method*), and some parameters. When such a call is answered by the receiver the local process of the sender is suspended until the execution of the specified method (by the receiver) has terminated, and the result has sent back.

Objects are grouped into *classes*. All objects in one class (the *instances* of that class) have the same kind of variables, the same methods, and the same local process. In this way a class describes the behaviour of its instances.

One of the main proof theoretical problems of such an object-oriented language is how to reason about dynamically evolving *pointer structures*. To master the complexity of this problem we investigated several approximations of the language POOL.

First we studied the proof theory of a version of POOL called P ([B1]). The objects of the systems described by this language may interact only by a

synchronous communication mechanism which consists of sending and receiving objects. When an object wants to send an object it explicitly states to which object. On the other hand receiving an object does not require in general the identification of the communication partner. In those cases then the communication partner is selected non-deterministically. This communication mechanism is similar to the one embodied in the language CSP. The main difference is that the communication partner in CSP is identified statically, which is not the case in the language P.

To describe a system of objects we view variables as *dynamic one-dimensional arrays*. The objects themselves are identified by integers in such a way that the value of a variable x of an object identified by the number n is given by the nth element of the array denoted by x. Using this scheme we showed how to apply the proof theory developed for CSP ([AFR]) to this language P.

However, this coding mechanism of objects makes the abstraction level of the reasoning about program correctness less high than that of the programming language. Pierre America developed a proof theory for the language SPOOL (a Sequential version of POOL) in which one reasons about a system of objects at a higher abstraction level ([A1]). In more detail, this means the following:

- The only operations on "pointers" (references to objects) are
  - testing for equality
  - dereferencing (looking at the value of a variable of the referenced object)
- In a given state of the system, it is only possible to mention the objects that exist in that state. Objects that do not (yet) exist never play a role.

Strictly speaking, direct dereferencing is not even allowed in the programming language, because each object only has access to its own [illegible] variables. But to dispense with this feature would ask for even more advanced techniques to reason about the correctness of a program.

The completeness proof of this proof system for SPOOL requires quite an elaborate use of the standard techniques ([Ba]), one might almost say that these techniques are, in this application, "stretched to their utmost limits".

This abstract way of reasoning about dynamically evolving pointer structures we then applied to the language P ([A2]). Described very briefly the resulting proof method consists of the following elements:

- A *local* stage. Here we deal with all statements that do not involve communication or object creation. These statements are proved correct with respect to pre- and postconditions in the usual manner of sequential programs [Ba,Ho]. At this stage, we just use *assumptions* to describe the behaviour of the communication and creation statements. These will be verified in the next stage. In this local stage, a *local assertion language* is used, which only talks about the current object in isolation.

- An *intermediate* stage. In this stage the above assumptions about communication and creation statements are verified. Here a *global assertion language* is used, which reasons about all the objects in the system. For each creation statement and for each pair of possibly communicating send and receive statements is verified that the specification used in the local proof system is consistent with the global behaviour.
- A *global* stage. Here some properties of the system as a whole can be derived from a kind of standard specification that arises from the intermediate stage. Again the global assertion language is used.

Finally we showed how to generalise the proof theory developed for the Ada rendez-vous ([G]) to the rendez-vous mechanism of POOL ([B2]). The main difference between these two mechanisms being that in the language Ada we have a statically fixed recursion depth of a rendez-vous, whereas in POOL we do not have such a static bound to the recursion depth.

Acknowledgement

I want to stress in particular Jaco's persistent insistence on a high quality of the presentation without which most of this work would be a mere solipsistic activity. Especially this field of proof theory, which gives rise to rather complicated formalisms, requires special care concerning the presentation.

References

[Am] P. America: *Definition of the programming language POOL-T*. ESPRIT project 415A, Doc. No. 0091, Philips Research Laboratories, Eindhoven, the Netherlands, September 1985.

[A1] P. America, F.S. de Boer: *A proof system for a sequential version of POOL*. C.W.I. Report, to appear.

[A2] P. America, F.S. de Boer: *A proof system for a parallel language with dynamic process creation*. In: Deliverable of the ESPRIT 415 Working Group on Semantics and Proof techniques, 1988.

[AFR] K.R. Apt, N. Francez, W.P. de Roever: *A proof system for communicating processes*. ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3, 1980, pp. 359–385.

[Ba] J.W. de Bakker: *Mathematical theory of program correctness*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.

[B1] F.S. de Boer: *A proof-rule for process-creation*. In: M. Wirsing (ed.): *Formal Description of Programming Concepts 3*, Proc. of the third IFIP WG 2.2. working conference, Gl. Avernaes, Ebberup, Denmark, August 25-28.

[B2] F.S. de Boer: *A proof system for the language POOL*. C.W.I. Report, to appear.

[G] R. Gerth, W.P. de Roever: *A proof system for concurrent Ada programs*. Science of Computer Programming 4, pp. 159-204.

[Ho] C.A.R. Hoare: *An axiomatic basis for computer programming*. Communications of the ACM, Vol. 12, No. 10, 1969, pp. 567–580,583.

Continuation Semantics for PROLOG with Cut

A. de Bruin

Economische Faculteit, Erasmus Universiteit
Postbus 1738, NL-3000 DR Rotterdam

E.P. de Vink

Faculteit Wiskunde & Informatica, Vrije Universiteit
De Boelelaan 1081, NL-1081 HV Amsterdam

ABSTRACT

We present an operational model $\mathcal{O}$ and a continuation based denotational model $\mathcal{D}$ for a uniform variant of Prolog, including the cut operator. The two semantical definitions make use of higher order transformations Φ and Ψ, respectively. We prove $\mathcal{O}$ and $\mathcal{D}$ equivalent by comparing yet another pair of higher order transformations $\tilde{\Phi}$ and $\tilde{\Psi}$, that yield Φ respectively Ψ by application of a suitable abstraction operator.

1. Introduction

In [BV] we presented both an operational and a denotational continuation based semantics for the core of PROLOG, and we proved these two semantics equivalent. We used a two step approach, by first deriving these results for an intermediate language, obtained by stripping the logic programming aspects (substitutions, most general unifiers and all that) from PROLOG. This resulted in the abstract language $\mathcal{B}$ in which only the control structures from PROLOG remained, such as the backtrack mechanism and the cut operator. After having compared the operational and denotational meanings for $\mathcal{B}$ successfully we generalized as a next step the two semantics to the case of PROLOG while preserving their equivalence.

The language $\mathcal{B}$ will be investigated again in this paper, but now more a a guinea pig. We will use it to test a new idea for proving equivalence of operational and denotational semantics based on cpo's. The main virtue of $\mathcal{B}$ in this respect is that although it is a sequential language it has a nontrivial control structure. In fact, the denotational semantics of this language needs three continuations to adequately describe the flow of control.

We will discuss our new approach to equivalence proofs by comparing it with the standard way these proofs have been conducted so far. To this end, we first spend a few words on operational semantics. The main idea behind this brand of semantics is to describe how an abstract machine executes a program in the language of interest. The abstract machine is defined by specifying the configurations it can be in, and by introducing a step function, mapping configurations to configurations, thus describing the behaviour of the abstract machine. Starting from an initial configuration C_0, repeatedly applying

the step function will deliver a number of intermediate configurations C_o, C_1, C_2, .. with $C_{i+1} = step\ C_i$. The computation terminates when a final configuration has been reached.

Our operational meaning function $\mathcal{O}$ abstracts from these intermediate results however, defining the meaning $\mathcal{O}[\![s]\!]$ of a statement s as a state transformation, mapping initial states to final states while recording the wanted observations. The final state is obtained by iterating the step function and this iterations can nicely be captured by taking a fixed point of a suitable higher order operator Φ.

Notice that, due to the abstraction of (most of) the intermediate configurations we made, in general the operator Φ will not have a unique fixed point. For instance, if iterating the step function fails to produces a final configuration, i.e. we deal with a nonterminating computation, and in that case we put $\mathcal{O}[\![s]\!] = \perp$. Other fixed points of Φ are possible, yielding different results for such s. The fact that fixed points are not unique complicates maters when it comes to prove $\mathcal{O}$ equivalent to $\mathcal{D}$, the denotational meaning function. The standard technique in such a proof is to show that a step of the abstract machine does not affect the denotational meaning of the configurations being transformed. More technically, if we have that $step\ C = C'$, and if we extend $\mathcal{D}$ to a function $\mathcal{I}$ taking configurations as arguments, then we have to show hat $\mathcal{I}C = \mathcal{I}C'$. From this result we will then be able to infer hat $\mathcal{I}C_{initial} = \mathcal{I}C_{final}$, by induction on the number of steps needed. We then deduce that ($) $\mathcal{D}[\![s]\!]\sigma = \mathcal{O}[\![s]\!]\sigma$.
However, there is a flaw in this line of reasoning. The result ($) is only valid for terminating computations. If iterating the step function does not produce a final configuration then the above argument does not work. This means that in order to complete the equivalence proof $\mathcal{D} = \mathcal{O}$, we have to derive the result that $\mathcal{O}[\![s]\!]\sigma = \perp$ implies $\mathcal{D}[\![s]\!]\sigma = \perp$. Unfortunately this takes at least as much effort as was needed to derive the previous result, see for example the proofs in [Ba1].

Now if the operator Φ would have a unique fixed point then it would not be necessary to derive this additional result. Uniqueness is guaranteed for instance when one does not use cpo's and a continuity argument to ensure the existence of the fixed points, but when complete metric spaces are used instead and the operators are contracting functions on these spaces. For in that case Banach's theorem can be applied. In [KR] unicity of the fixed point of the operational higher order operator Φ has been exploited to derive compact equivalence proofs for operational and denotational meanings along the lines sketched above. A similar line of reasoning has been used in [BM]. The fact that our operator Φ admits more than one fixed point seems to be an essential consequence due to the fact that we abstracted away from the intermediate configurations. For instance, it is not possible to define in a straightforward way a contraction on metric spaces which yield the same operational semantics. The idea behind our equivalence proof is to try to take best of both worlds.

We introduce a slightly less abstract operational semantics using a new operator $\bar{\Phi}$ that does have a unique fixed point. Our semantics is made more concrete because it does not simply deliver observables σ as the result of a computation, but also additional information. Outputted states σ are preceded by a number of clock ticks, a row of τ's. The idea is that each τ in this row corresponds with the execution of an elementary action by the abstract machine, i.e. one iteration of the step function. Similarly for a nonterminating computation, we do not deliver $\perp$ but an infinite row of τ's instead. So $\perp$ is reformulated as internal divergence. Now, for the corresponding meaning functions $\bar{\mathcal{O}}$ and $\bar{\mathcal{D}}$ a compact equivalence proof can be given. In order to establish from this the equivalence of our original functions $\mathcal{O}$ and $\mathcal{D}$ it is sufficient to show that, apart from proving $\bar{\mathcal{O}} = \bar{\mathcal{D}}$, there exists an abstraction operator ρ, "a τ-remover" so to speak, such that for all s and σ we have (*) $\mathcal{O}[\![s]\!]\sigma = \rho(\bar{\mathcal{O}}[\![s]\!]\sigma)$ and

$\mathcal{D}[\![s]\!] \sigma = \rho(\tilde{\mathcal{D}}[\![s]\!] \sigma)$.

In the next section we will show that this idea can be made to work for a simple language, the most complicated construct of which is the while statement. However it will also become clear that the complexity of the over all proof has not diminished. This is caused by the fact that substantial additional work ahs to be performed in proving the equalities (*) above. On the other hand, the reasoning in these proofs is to a great extent independent of the particular language investigated. The result on the operational semantics is valid for each operational semantics derived from (deterministic) step functions and in the proof of the other result in (*) a number a generic elements seems to be present as well, which can be carried over unaltered to similar proofs for other languages. These observations are worked out in section 3, where we present a more general theory on the relation between abstract domains and concretizations thereof like the ones discussed above. We show that the abstract domains can be considered as so called retracts of the more concrete ones. We will derive a few theorems, related to those of [BMZ] and [Me] that enable us to prove results as in (*) in a more smooth and elegant way. In the remaining sections this theory is tested using the above described language $\mathcal{B}$: Section 4 describes the operational and denotational semantics of $\mathcal{B}$. Section 5 will be devoted to the actual equivalence proof.

Acknowledgements. From the above it will be clear that our work relies heavily on that of others. The immediate starting point of this paper is [KR] where for the first time the observation was made that compact equivalence proofs could be realized using higher order transformations. (But see also [BBKM].) We most notably benefit from the work on metric semantics of concurrency perform by De Bakker e.a. E.g. [BZ], [BKMOZ], [BM], [Ba2]. It is a pleasure to thank the forum formed by the members of the working group on concurrency, - Jaco de Bakker, Frank de Boer, Joost Kok, Jan Rutten and others - for their comments and the good scientific atmosphere they provided. Finally we are grateful to M279 for her hospitality.

2. A Simple Example: the While Statement

In this section we will illustrate the basic idea behind our new equivalence proof, by sketching how such a proof can be given for the very simple language defined below.

(2.1) DEFINITION The set of elementary statements *EStat* is defined by $e ::= x{:=}t \mid$ *if* b *then* s_1 *else* s_2 *fi* $\mid$ *while* b *do* s *od*. The set of statements *Stat* is defined by $s ::= \varepsilon \mid e;s$.

As remarked in section 1, an operational semantics is defined via an abstract machine. Such a machine, called a transition system, can be specified by giving its configurations together with a relation between configurations which describes the step function. To avoid too many details we assume the existence of an interpretation I from which the effect of executing an assignment statement can be obtained, as well as the value of boolean expressions. Notice that there is no transition defined from configurations of the form $[\varepsilon, \sigma]$. These are the final configurations corresponding to terminated

computations.

(2.2) DEFINITION

(i) The set of configurations *Conf* is defined as the collection of statement-state pairs $[s,\sigma]$, i.e. $Conf = \{ [s,\sigma] \mid s \in Stat, \sigma \in \Sigma \}$.

(ii) The step function $\rightarrow$ is the smallest subset of $Conf \times Conf$ s.t.

$[x{:=}t,\sigma] \rightarrow [s,\sigma']$ where $\sigma' = I[\![x{:=}t]\!]\sigma$

$[\mathit{if}\ b\ \mathit{then}\ s_1\ \mathit{else}\ s_2\ \mathit{fi};s,\sigma] \rightarrow [s_1;s,\sigma]$ if $I[\![b]\!]\sigma = tt$

$[\mathit{if}\ b\ \mathit{then}\ s_1\ \mathit{else}\ s_2\ \mathit{fi};s,\sigma] \rightarrow [s_2;s,\sigma]$ if $I[\![b]\!]\sigma = ff$

$[\mathit{while}\ b\ \mathit{do}\ s'\ \mathit{od};s,\sigma] \rightarrow [s';\mathit{while}\ b\ \mathit{do}\ s'\ \mathit{od};s,\sigma]$ if $I[\![b]\!]\sigma = tt$

$[\mathit{while}\ b\ \mathit{do}\ s'\ \mathit{od};s,\sigma] \rightarrow [s,\sigma]$ if $I[\![b]\!]\sigma = ff$

(iii) The operational semantics $\mathcal{O} : Stat \rightarrow \Sigma \rightarrow \Sigma_\perp$ is defined by $\mathcal{O}[\![s]\!]\sigma = \mu\Phi([s,\sigma])$ where $\Phi : [(Conf \rightarrow \Sigma_\perp) \rightarrow (Conf \rightarrow \Sigma_\perp)]$ is given by $\Phi O[\varepsilon,\sigma] = \sigma$, $\Phi OC = OC'$ if $C \rightarrow C'$, and $\Sigma_\perp$ denotes the flat cpo with least element $\perp$.

Our denotational semantics will use continuations. Although this seems to be a bit too heavy for such a simple language, we do not use direct semantics for two reasons. First of all, the equivalence proof will proceed more smoothly when using continuations and secondly, the language $\mathcal{B}$ cannot be given a satisfactory direct semantics. (In order to model the cut operator this way one has to resort to cut-flags or other kinds of indicators. See [JM], [DM], [Bd], [Vi].) Since this section is intended as an introduction, we want the two equivalence proofs for $\mathcal{B}$ and the simple language to be as similar as possible.

Our meaning function $\mathcal{D}$ is defined as the least fixed point of a higher order operator Ψ. In fact, it does not matter much how $\mathcal{D}$ is defined (as long as it remains denotational), the more usual approach based on environments as in [Ba1] would work equally well, cf. [BM]. The style of defining here is closer to [KR] which we take as a starting point.

(2.3) DEFINITION

(i) Domains: The collections *Cont* of continuations and the set *Meaning* of meanings are given by $Cont \rightarrow \Sigma \rightarrow \Sigma_\perp$ and $Meaning = Stat \rightarrow [Cont \rightarrow \Sigma \rightarrow \Sigma_\perp]$.

(ii) Functions: The denotational semantics $\mathcal{D} : Stat \rightarrow \Sigma \rightarrow \Sigma_\perp$ is defined by $\mathcal{D}[\![s]\!]\sigma = \mu\Psi[\![s]\!]\xi_o\sigma$ where $\xi_o = \lambda\sigma.\sigma$ and $\Psi : [Meaning \rightarrow Meaning]$ is given by

$\Psi M[\![\varepsilon]\!]\xi\sigma = \xi\sigma$,

$\Psi M[\![x{:=}t]\!]\xi\sigma = \xi\sigma'$ if $\sigma' = I[\![x{:=}t]\!]\sigma$

$\Psi M[\![\mathit{if}\ b\ \mathit{then}\ s_1\ \mathit{else}\ s_2\ \mathit{fi}]\!]\xi\sigma = M[\![s_1]\!]\xi\sigma$ if $I[\![b]\!]\sigma = tt$

$\Psi M[\![\mathit{if}\ b\ \mathit{then}\ s_1\ \mathit{else}\ s_2\ \mathit{fi}]\!]\xi\sigma = M[\![s_2]\!]\xi\sigma$ if $I[\![b]\!]\sigma = ff$

$\Psi M[\![\mathit{while}\ b\ \mathit{do}\ s\ \mathit{od}]\!]\xi\sigma = M[\![s]\!]\{ M[\![\mathit{while}\ b\ \mathit{do}\ s\ \mathit{od}]\!]\xi \}\sigma$ if $I[\![b]\!]\sigma = tt$

$\Psi M[\![\mathit{while}\ b\ \mathit{do}\ s\ \mathit{od}]\!]\xi\sigma = \xi\sigma$ if $I[\![b]\!]\sigma = ff$

$\Psi M[\![e;s]\!]\xi\sigma = M[\![e]\!]\{ M[\![s]\!]\xi \}\sigma$.

Before giving the present equivalence proof, we discuss the old approach for a while. First of all one proves $\mathcal{O} \leq \mathcal{D}$. The idea is to extend $\mathcal{D}$ to a intermediate function $\mathcal{I}$ defined on configurations: $\mathcal{I}[s,\sigma] = \mathcal{D}[\![s]\!]\sigma$. One then proves $\Phi\mathcal{I} = \mathcal{I}$, in essence by checking this for all possible

configurations (cf. 2.2.ii, and the proof of 2.7). From this result $\mathcal{O} \le \mathcal{D}$ follows immediately, but in fact we have more than that. Because $\mu\Phi$ delivers results in $\Sigma_\perp$, a flat domain, we have that for all s and σ such that $\mu\Phi[\,s,\sigma\,] = \sigma' \neq \perp$, also $\mathcal{I}[\,s,\sigma\,] = \sigma'$ holds. Therefore, in order to complete our proof we need to show $\mathcal{O}[\![\,s\,]\!]\sigma = \perp \Rightarrow \mathcal{D}[\![\,s\,]\!]\sigma = \perp$. This is most easily accomplished by showing that for all approximations $\Psi^i\perp$ of $\mathcal{D}$ we have $\Psi^i\perp \le \mathcal{O}$, and this can be proved by induction on i, checking all possible forms a statement s can take.

Now this second half of the equivalence proof would not be needed if Φ would have a unique fixed point: for $\Phi\mathcal{I} = \mathcal{I}$ would then imply $\mathcal{I} = \mu\Phi$ and thus $\mathcal{O} = \mathcal{D}$. The idea now is to make Φ a little bit more concrete, making it deliver result in the cpo $\bar{\Sigma}^{st}$ of streams instead of the flat domain $\Sigma_\perp$. This will be accomplished by prefixing a result σ' with a number of τ's, each of which denotes a "clock tick", corresponding with an elementary step of our abstract machine. The effect of all this will be that our new operator $\bar{\Phi}$ will indeed have a unique fixed point (cf. lemma 2.5).

(2.4) DEFINITION

(i) Put $\bar{\Sigma} = \Sigma \cup \{\tau\}$ for some distinguished $\tau \notin \Sigma$. $\bar{\Sigma}$ is ranged over by θ. Let $\bar{\Sigma}$ denote the cpo of streams over $\bar{\Sigma}$, cf. [Me], [MV].

(ii) The step function $\rightarrow$ is the smallest subset of $Conf \times \bar{\Sigma} \times Conf$ s.t.

$[\,x{:=}t,\sigma\,] \rightarrow_\tau [\,s,\sigma'\,]$ where $\sigma' = I[\![\,x{:=}t\,]\!]\sigma$

$[\,if\ b\ then\ s_1\ else\ s_2\,fi;s,\sigma\,] \rightarrow_\tau [\,s_1;s,\sigma\,]$ if $I[\![\,b\,]\!]\sigma = tt$

$[\,if\ b\ then\ s_1\ else\ s_2\,fi;s,\sigma\,] \rightarrow_\tau [\,s_2;s,\sigma\,]$ if $I[\![\,b\,]\!]\sigma = ff$

$[\,while\ b\ do\ s'\ od;s,\sigma\,] \rightarrow_\tau [\,s';while\ b\ do\ s'\ od;s,\sigma\,]$ if $I[\![\,b\,]\!]\sigma = tt$

$[\,while\ b\ do\ s'\ od;s,\sigma\,] \rightarrow_\tau [\,s,\sigma\,]$ if $I[\![\,b\,]\!]\sigma = ff$

(iii) The operational semantics $\bar{\mathcal{O}} : Stat \rightarrow \Sigma \rightarrow \bar{\Sigma}^{st}$ is defined by $\bar{\mathcal{O}}[\![\,s\,]\!]\sigma = \mu\bar{\Phi}([\,s,\sigma\,])$ where $\bar{\Phi} \in [\,(Conf \rightarrow \bar{\Sigma}^{st}) \rightarrow (Conf \rightarrow \bar{\Sigma}^{st})\,]$ is given by $\bar{\Phi}O[\,\varepsilon,\sigma\,] = \sigma$, $\bar{\Phi}OC = \tau \cdot OC'$ if $C \rightarrow C'$.

Notice that definitions 2.2(ii) and 2.4(ii) are exactly the same except for the labels τ.

It is instructive to observe the relation between the functions $\mathcal{O}$ and $\bar{\mathcal{O}}$. We observe, without proof that $\mathcal{O}[\![\,s\,]\!]\sigma = \sigma' \Leftrightarrow \exists k \in \mathbb{N}{:}\bar{\mathcal{O}}[\![\,s\,]\!]\sigma = \tau^k.\sigma'$ and $\mathcal{O}[\![\,s\,]\!]\sigma = \perp \Leftrightarrow \bar{\mathcal{O}}[\![\,s\,]\!]\sigma = \tau^\omega$. Of course a similar result is true for $\mu\Phi$ and $\mu\bar{\Phi}$. Notice that this implies that for all configurations C we have that $\mu\Phi C$ is maximal in $\bar{\Sigma}^{st}$ and this again means that $\mu\bar{\Phi}$ itself is maximal. Therefore $\mu\Phi$ is not only the least fixed point of $\bar{\Phi}$; it is the only one!

(2.5) LEMMA *$\bar{\Phi}$ has a unique fixed point.*

PROOF We prove hat $\mu\bar{\Phi}$ is maximal, from which it follows hat $\mu\bar{\Phi}$ is unique. The proof is by contradiction. Suppose $\mu\bar{\Phi}$ is not maximal. Then there exists at least one C with the property that $\mu\bar{\Phi}C$ is not maximal in $\bar{\Sigma}^{st}$, i.e. $\mu\bar{\Phi}C$ must be of the form $x.\perp$. Now choose from the set of all configurations with this property one configuration, say $\bar{C}$, such that $\mu\bar{\Phi}\bar{C}$ has minimal length.
Because $\mu\bar{\Phi}$ is a fixed point, we have $\mu\bar{\Phi}C = \bar{\Phi}(\mu\bar{\Phi}C) = \tau.\mu\bar{\Phi}C'$ for some $C' \in Conf$ s.t. $C \rightarrow C'$. This however means that $\mu\bar{\Phi}C'$ is also of the form $x.\perp$, which contradicts the minimality of $|\ \mu\bar{\Phi}\bar{C}\,|$.
□

We now have to define a denotational semantics $\bar{\mathcal{D}}$ which should be equivalent with $\bar{\mathcal{O}}$. This is

done below. Notice that some care has to be taken, where τ's are added in the defining clauses of $\tilde{\Psi}$ as compared to the definition of Ψ in 2.3. (Notice furthermore that, although our notation does not show this, the standard continuation ξ_o now delivers a one element stream from $\tilde{\Sigma}^{st}$, whereas in definition 2.3 a single element in $\Sigma_\perp$ was delivered. We tacitly consider $\Sigma_\perp$ as a subcpo of $\tilde{\Sigma}^{st}$.)

(2.6) DEFINITION

(i) Domains: The collections $Cont^{\sim}$ of continuations and the set $Meaning^{\sim}$ of meanings are given by $Cont^{\sim} \to \Sigma \to \tilde{\Sigma}^{st}$ and $Meaning^{\sim} = Stat \to [\, Cont^{\sim} \to \Sigma \to \tilde{\Sigma}^{st}\,]$.

(ii) The denotational semantics $\tilde{\mathcal{D}} : Stat \to \Sigma \to \tilde{\Sigma}^{st}$ is defined by $\tilde{\mathcal{D}}[\![\, s \,]\!]\sigma = \mu\tilde{\Psi}[\![\, s \,]\!]\xi_o\sigma$ where $\xi_o = \lambda\sigma.\sigma$ and $\tilde{\Psi} : [\, Meaning^{\sim} \to Meaning^{\sim}\,]$ is given by

$\tilde{\Psi}M[\![\, \varepsilon \,]\!]\xi\sigma = \xi\sigma$,

$\tilde{\Psi}M[\![\, x{:=}t \,]\!]\xi\sigma = \tau\bullet\xi\sigma'$ if $\sigma' = I[\![\, x{:=}t \,]\!]\sigma$

$\tilde{\Psi}M[\![\, if\ b\ then\ s_1\ else\ s_2 fi \,]\!]\xi\sigma = \tau\bullet M[\![\, s_1 \,]\!]\xi\sigma$ if $I[\![\, b \,]\!]\sigma = tt$

$\tilde{\Psi}M[\![\, if\ b\ then\ s_1\ else\ s_2 fi \,]\!]\xi\sigma = \tau\bullet M[\![\, s_2 \,]\!]\xi\sigma$ if $I[\![\, b \,]\!]\sigma = ff$

$\tilde{\Psi}M[\![\, while\ b\ do\ s\ od \,]\!]\xi\sigma = \tau\bullet M[\![\, s \,]\!]\{\, M[\![\, while\ b\ do\ s\ od \,]\!]\xi \,\}\sigma$ if $I[\![\, b \,]\!]\sigma = tt$

$\tilde{\Psi}M[\![\, while\ b\ do\ s\ od \,]\!]\xi\sigma = \tau\bullet\xi\sigma$ if $I[\![\, b \,]\!]\sigma = ff$

$\tilde{\Psi}M[\![\, e;s \,]\!]\xi\sigma = M[\![\, e \,]\!]\{\, M[\![\, s \,]\!]\xi \,\}\sigma$.

After extending $\tilde{\mathcal{D}}$ to the intermediate function $\tilde{\mathcal{F}}$ acting on configurations we have the following main lemma.

(2.7) LEMMA *Define* $\tilde{\mathcal{F}} : Conf \to \tilde{\Sigma}^{st}$ *by* $\tilde{\mathcal{F}}([\, s,\sigma\,]) = \tilde{\mathcal{D}}[\![\, s \,]\!]\sigma$. *Then it holds that* $\tilde{\Phi}\tilde{\mathcal{F}} = \tilde{\mathcal{F}}$.

PROOF We have to prove $\tilde{\Phi}\tilde{\mathcal{F}}C = \tilde{\mathcal{F}}C$ for all configurations C. We only consider the case $C = [\, while\ b\ do\ s'od;s,\sigma\,]$ with $I[\![\, b \,]\!]\sigma = tt$. Then on the one hand $\tilde{\Phi}\tilde{\mathcal{F}}C = \tau\bullet\tilde{\mathcal{F}}[\, s'{:}while\ b\ do\ s'\ od;s,\sigma\,] = \tau\bullet\tilde{\mathcal{D}}[\![\, s'{:}while\ b\ do\ s'\ od;s \,]\!]\xi_o\sigma$, and on the other $\tilde{\mathcal{F}}C = \tilde{\mathcal{D}}[\![\, while\ b\ do\ s'\ od;s \,]\!]\xi_o\sigma = \tilde{\mathcal{D}}[\![\, while\ b\ do\ s'\ od \,]\!]\{\, \tilde{\mathcal{D}}[\![\, s \,]\!]\xi_o \,\}\sigma = \tau\bullet\tilde{\mathcal{D}}[\![\, s' \,]\!]\{\, \tilde{\mathcal{D}}[\![\, while\ b\ do\ s'\ od \,]\!]\{\, \tilde{\mathcal{D}}[\![\, s \,]\!]\xi_o \,\}\,\}\sigma = \tau\bullet\tilde{\mathcal{D}}[\![\, s' \,]\!]\{\, \tilde{\mathcal{D}}[\![\, while\ b\ do\ s'\ od;s \,]\!]\xi_o \,\}\sigma = \tau\bullet\tilde{\mathcal{D}}[\![\, s'{:}while\ b\ do\ s'\ od;s \,]\!]\xi_o\sigma$. □

This lemma and lemma 2.5 establishes the first part of our proof, viz. $\tilde{\mathcal{O}} = \tilde{\mathcal{D}}$ since this follows from $\mu\tilde{\Phi} = \mathcal{F}$. In order to derive $\mathcal{O} = \mathcal{D}$ we introduce an operator which removes the τ's from the results of $\tilde{\mathcal{O}}$ and $\tilde{\mathcal{D}}$, cf. the remarks following definition 2.4. This abstraction operator *strip* is defined as follows.

(2.8) DEFINITION The function $strip \in [\, \tilde{\Sigma}^{st} \to \Sigma_\perp \,]$ is defined by $strip = \mu\mathrm{P}$ where $\mathrm{P} \in [\, [\, \tilde{\Sigma}^{st} \to \Sigma_\perp \,] \to [\, \tilde{\Sigma}^{st} \to \Sigma_\perp \,]\,]$ is defined by $\mathrm{P}\rho\perp = \perp$, $\mathrm{P}\rho\varepsilon = \varepsilon$, $\mathrm{P}\rho(\tau.x) = \rho x$, $\mathrm{P}\rho(\sigma.x) = \sigma$.

So *strip* yields the first proper state of a stream over $\tilde{\Sigma}$. Notice that this operator P indeed has the functionality as claimed, and that therefore *strip* is continuous. The next two lemma's now furnish the last results needed to prove $\mathcal{O} = \mathcal{D}$.

(2.9) LEMMA *For all* s *and* σ: $\mathcal{O}[\![\, s \,]\!]\sigma = strip(\tilde{\mathcal{O}}[\![\, s \,]\!]\sigma)$.

PROOF The same result holds for all approximations of $\mathcal{O}$ and $\tilde{\mathcal{O}}$, i.e. we have $\Phi^i \bot = strip(\bar{\Phi}^i \bot)$. This can be proved by induction on i. The lemma no follows from the continuity of *strip*. □

(2.10) LEMMA *For all s and* σ: $\mathcal{D}[\![s]\!] \sigma = strip(\tilde{\mathcal{D}}[\![s]\!] \sigma)$.

PROOF We first prove a somewhat stronger fact about the approximations: If for some $\xi \in Cont$, $\bar{\xi} \in Cont^{-}$ we have $\xi = strip \circ \bar{\xi}$, then for all i, s and σ: $\Psi^i \bot [\![s]\!] \xi\sigma = strip(\bar{\Psi}^i \bot [\![s]\!] \bar{\xi}\sigma)$. This fact can be proven by induction on i, checking all possibilities for s. As an example we consider the statement *while b do s od*, evaluated in a state in which b is true. $\Psi^i \bot [\![\textit{while b do s od}]\!] \xi\sigma = \Psi^{i-1} \bot [\![s]\!] \{ \Psi^{i-1} \bot [\![\textit{while b do s od}]\!] \xi \} \sigma$. Now from the induction hypothesis we learn that $\Psi^{i-1} \bot [\![\textit{while b do s od}]\!] \xi = strip(\bar{\Psi}^{i-1} \bot [\![\textit{while b do s od}]\!] \bar{\xi})$ and applying the induction hypothesis again, using this fact we infer $\Psi^{i-1} \bot [\![s]\!] \{ \Psi^{i-1} \bot [\![\textit{while b do s od}]\!] \xi \} = strip(\tau \bullet \bar{\Psi}^i \bot [\![\textit{while b do s od}]\!] \bar{\xi}\sigma)$. □

From the above lemmas and lemma 2.7 we derive as outlined before the equivalence of the operational and denotational semantics for $\mathcal{B}$.

(2.11) THEOREM $\mathcal{O} = \mathcal{D}$. □

Let us compare the new equivalence proof with the older one discussed after definition 2.3. Indeed, the core of our proof (lemma 2.7) is more compact now, but we had to pay a price: lemmas 2.9 and 2.10 had to be proven as well. For lemma 2.9 this is no big problem though. This proof does not depend on the underlying language, but only on the way the operator Φ has been derived from the transition system. Therefore this lemma can be used for other languages as well, (cf. section 5), it needs to be proven only once.

The proof of lemma 2.10 seems to depend more on the underlying language. At first sight it looks like the work we disposed of in lemma 2.7 now bounces back at us. However in this proof as well there is a language independent part. In order to see this it is worthwhile to study the relation between definition 2.6 and 2.3. In the latter one we inserted τ's while in the former one we do not have such clock ticks. Notice however that if we omit the τ's from definition 2.6 we get definition 2.3 back. Notice also that there is a similar relation between definitions 2.4 and 2.2. With some abuse of notation this relation can be written as (2.2) = *strip*(2.4) and (2.3) = *strip*(2.6).

Now these definitions established operators Φ, Ψ, $\bar{\Phi}$ and $\bar{\Psi}$, and by taking least fixed points we derived at the functions $\mathcal{O}$, $\mathcal{D}$, $\tilde{\mathcal{O}}$ and $\tilde{\mathcal{D}}$. For these resultant functions we have proven a similar result as claimed for the definitions: $\mathcal{O} = strip(\tilde{\mathcal{O}})$ and $\mathcal{D} = strip(\tilde{\mathcal{D}})$, (again with some abuse of notation). We would like to have a generic theorem that would provide us with the above relations in one blow: Let $\bar{\Delta}$ be a definition of some higher order operator $\bar{\Theta}$ and let Δ be the same definition, only without τ's, defining an operator Θ, (i.e. $\Delta = strip(\bar{\Delta})$. Then, under certain restrictions on the form of $\bar{\Delta}$ we have $\mu\Theta = strip(\mu\bar{\Theta})$.

In the next sections we will develop some theory in which these ideas are worked out.

3. Retractions

In this section we develop a little theory about pairs of cpo's of which can be consider less abstract than the other. We will give sufficient conditions under which the least fixed point of a transformation maps on the least fixed point of its abstract version.

(3.1) DEFINITION Let $D, \tilde{D}$ be cpo's. D is called a retract of $\tilde{D}$ if there exist two continuous mappings $i: D \rightarrow \tilde{D}$, $j: \tilde{D} \rightarrow D$ s.t. $j \circ i = id_D$.

We write in the above situation $D \leq_{i,j} \tilde{D}$ or just $D \leq \tilde{D}$. If $D \leq_{i,j} \tilde{D}$ then i is an embedding and j is strict. (Injectivity of i follows directly from $j \circ i = id_D$; strictness of j follows from $j(\bot_{\tilde{D}}) \leq_D j(i(\bot_D)) = \bot_D$.) In the context of $D \leq_{i,j} \tilde{D}$ we call i the inclusion and j the retraction, respectively.

Consider the cpo's $D = \Sigma_\bot$, $\tilde{D} = \tilde{\Sigma}^{st}$ augmented with the stream ordering. Let $i: D \rightarrow \tilde{D}$ be the inclusion and let $j: \tilde{D} \rightarrow D$ be defined by $j = strip$, cf. section 2. Then we have $j \circ i(\bot) = strip(\bot) = \bot$ and $j \circ i(\sigma) = strip(\sigma.\varepsilon) = \sigma$. So $D \leq_{i,j} \tilde{D}$, i.e. $\Sigma_\bot \leq_{i,strip} \tilde{\Sigma}^{st}$.

The relation $\leq_{i,j}$ between cpo's is - roughly speaking - one half of the subdomain ordering in the category CPO. For the subdomain ordering there is the additional requirement that $i \circ j \leq id_{\tilde{D}}$. See [Pl].

Given cpo's $D, \tilde{D}$ the pair of continuous functions i,j s.t. $D \leq_{i,j} \tilde{D}$ is not unique. It is already the case (contrary to the subdomain ordering) that for fixed inclusion i there exist several retractions j s.t. $D \leq_{i,j} \tilde{D}$. For example, take $D = \tilde{D} = \mathbf{N} \cup \{\infty\}$ together with the standard ordering. Define $i: D \rightarrow \tilde{D}$ by $i(d) = 2d$. Define $j_1, j_2: \tilde{D} \rightarrow D$ by $j_1(\tilde{d}) = \lfloor \tilde{d}/2 \rfloor$ and $j_2(\tilde{d}) = \lceil \tilde{d}/2 \rceil$. Clearly both j_1 and j_2 are continuous and satisfy $j_1 \circ i = id_D$, $j_2 \circ i = id_D$.

(3.2) DEFINITION Suppose $D \leq_{i,j} \tilde{D}$, $E \leq_{k,l} \tilde{E}$. A mapping $\tilde{\phi}: \tilde{D} \rightarrow \tilde{E}$ is called canonical if there exists $\phi: D \rightarrow E$ such that $l \circ \tilde{\phi} = \phi \circ j$. We define the function space $\tilde{D} \rightsquigarrow \tilde{E}$ of canonical mappings from $\tilde{D}$ to $\tilde{E}$ by $\tilde{D} \rightsquigarrow \tilde{E} = \{ \tilde{\phi}: \tilde{D} \rightarrow \tilde{E} \mid \tilde{\phi} \text{ canonical} \}$.

If $\tilde{\phi}: \tilde{D} \rightarrow \tilde{E}$ is canonical, then there exists a unique $\phi: D \rightarrow E$ s.t. $l \circ \tilde{\phi} = \phi \circ j$. For if $\phi_1, \phi_2: D \rightarrow E$ with $l \circ \tilde{\phi} = \phi_1 \circ j = \phi_2 \circ j$ then we have $\phi_1 = \phi_1 \circ j \circ i = l \circ \tilde{\phi} \circ i = \phi_2 \circ j \circ i = \phi_2$.

If D is a retract of $\tilde{D}$, say $D \leq_{i,j} \tilde{D}$, then we have an equivalence relation $\sim_D$ on $\tilde{D}$ (induced by j) defined by $d \sim_D d' \Leftrightarrow j(d) = j(d')$. For $\tilde{\phi}: \tilde{D} \rightarrow \tilde{E}$ we reformulate canonicity in terms of the equivalence relations $\sim_D$ and $\sim_E$. We will have the equivalence of (i) $\tilde{\phi}: \tilde{D} \rightarrow \tilde{E}$ is canonical and (ii) the induced mapping on equivalence classes $\tilde{\phi}: \tilde{D}/\sim_D \rightarrow \tilde{E}/\sim_E$ is well defined.

(3.3) LEMMA *Suppose* $D \leq_{i,j} \tilde{D}$, $E \leq_{k,l} \tilde{E}$. *Then it holds that* $\tilde{\phi}: \tilde{D} \rightarrow \tilde{E}$ *is canonical* $\Leftrightarrow \forall d,d' \in \tilde{D}$: $d \sim_D d' \Rightarrow \tilde{\phi}(d) \sim_D \tilde{\phi}(d')$.

PROOF "$\Rightarrow$" Choose $\phi: D \rightarrow E$ s.t. $l \circ \tilde{\phi} = \phi \circ j$. Let $d,d' \in \tilde{D}$ s.t. $d \sim_D d'$, i.e. $j(d) = j(d')$.

Then we have $l(\tilde{\phi}(d)) = \phi(j(d)) = \phi(j(d')) = l(\tilde{\phi}(d'))$, so $\tilde{\phi}(d) \sim_E \tilde{\phi}(d')$.
"$\Leftarrow$" Define $\phi : D \to E$ by $\phi = l \circ \tilde{\phi} \circ i$. Let $d \in \tilde{D}$. Then we have $\tilde{\phi}(i(j(d))) \sim_E \tilde{\phi}(d)$ and $\phi(j(d)) = l(\tilde{\phi}(i(j(d)))) = l(\tilde{\phi}(d))$, since $i(j(d)) \sim_D d$. Conclusion: $l \circ \tilde{\phi} = \phi \circ j$. □

The above lemma is not very deep but is is helpful in proving that $\tilde{D} \dashrightarrow \tilde{E}$ is a subcpo of the function space $\tilde{D} \to \tilde{E}$, since in the presence of 3.3 the proof takes places "in the world of $\tilde{D}$ and $\tilde{E}$."

(3.4) LEMMA *Suppose* $D \le_{i,j} \tilde{D}$, $E \le_{k,l} \tilde{E}$. *Then* $\tilde{D} \dashrightarrow \tilde{E}$ *is a cpo.*

PROOF Sufficient to prove: for a chain $\langle \phi_n \rangle_n$ in $\tilde{D} \dashrightarrow \tilde{E}$ is $\phi = lub_n\, \phi_n$ canonical. Suppose $d \sim_D d'$. Then by continuity of l and canonicity of ϕ_n: $l(\phi(d)) = l(\, lub_n\, \phi_n(d)) = lub_n\, l(\phi_n(d)) = lub_n\, l(\phi_n(d')) = l(\, lub_n\, \phi_n(d')) = l(\phi(d'))$, so $\phi(d) \sim_E \phi(d')$. □

Suppose D and E are retracts of $\tilde{D}$ and $\tilde{E}$, respectively. The function space $D \to E$ then, will be a retract of the function space $\tilde{D} \to \tilde{E}$. More precisely, if $D \le_{i,j} \tilde{D}$ and $E \le_{k,l} \tilde{E}$ then $(D \to E) \le_{I,J} (\tilde{D} \to \tilde{E})$ where $I = \lambda\phi.k \circ \phi \circ j$ and $J = \lambda\tilde{\phi}.l \circ \tilde{\phi} \circ i$. Continuity of I,J follows from the continuity of i through l. Furthermore $J \circ I = \lambda\phi.l \circ k \circ \phi \circ j \circ i = \lambda\phi.id_E \circ \phi \circ id_D = id_{D \to E}$. Analogously, if V is a set of values and $D \le_{i,j} \tilde{D}$ then $V \to D \le_{I,J} V \to \tilde{D}$ where $I = \lambda\phi.i \circ \phi$ and $J = \lambda\tilde{\phi}.j \circ \tilde{\phi}$.

Notice, for $\phi : D \to E$ is $I(\phi) : \tilde{D} \to \tilde{E}$ canonical: if $d \sim_D d'$ then $I(\phi)(d) = k(\phi(j(d))) = k(\phi(j(d'))) = I(\phi)(d')$ and a fortiori $I(\phi)(d) \sim_E I(\phi)(d')$. So we have $D \to E \le \tilde{D} \dashrightarrow \tilde{E}$. Moreover I and J preserve continuity, i.e. $\phi \in [D \to E] \Rightarrow I(\phi) \in [\tilde{D} \to \tilde{E}]$ and $\tilde{\phi} \in [\tilde{D} \to \tilde{E}] \Rightarrow J(\tilde{\phi}) \in [D \to E]$. Therefore we have $[D \to E] \le [\tilde{D} \to \tilde{E}]$. Combination of this two facts yields $[D \dashrightarrow E] \le [\tilde{D} \dashrightarrow \tilde{E}]$.

The notion of a retract was introduced here with the comparison of fixed points of higher order transformations in mind. By virtue of theorem 3.6 below it would therefore be convenient to have available a means for checking canonicity of these (higher order) transformations.

(3.5) LEMMA

(i) Choose cpo's $D \le_{i,j} \tilde{D}$, $E \le_{k,l} \tilde{E}$. Fix $\Phi : (\tilde{D} \to \tilde{E}) \to (\tilde{D} \to \tilde{E})$. Then it holds that $\Phi : (\tilde{D} \dashrightarrow \tilde{E}) \dashrightarrow (\tilde{D} \dashrightarrow \tilde{E}) \Leftrightarrow \forall\phi,\phi' \in \tilde{D} \dashrightarrow \tilde{E}\ \ \forall d,d' \in \tilde{D}:\ \phi \sim_{D \to E} \phi' \wedge d \sim_D d' \Rightarrow \Phi(\phi)(d) \sim_E \Phi(\phi')(d')$.

(ii) Let V be a set, $D \le_{i,j} \tilde{D}$ and fix $\Phi : (V \to \tilde{D}) \to (V \to \tilde{D})$. Then it holds that $\Phi : (V \to \tilde{D}) \dashrightarrow (V \to \tilde{D}) \Leftrightarrow \forall\phi,\phi' \in \tilde{D} \dashrightarrow \tilde{E}\ \forall v \in V : \phi \sim_{V \to D} \phi' \Rightarrow \Phi(\phi)(v) \sim_E \Phi(\phi')(v)$.

PROOF We only check (i). For $\tilde{\phi},\psi \in \tilde{D} \dashrightarrow \tilde{E}$ we have $\tilde{\phi} \sim_{D \to E} \psi \Leftrightarrow [\, d \sim_D d' \Rightarrow \tilde{\phi}(d) \sim_E \psi(d')\,]$. For it holds that $\tilde{\phi} \sim_{D \to E} \psi \Leftrightarrow l \circ \tilde{\phi} \circ i \Leftrightarrow l \circ \psi \circ i \Leftrightarrow \forall d \in D : l(\tilde{\phi}(i(d))) = l(\psi(i(d))) \Leftrightarrow \forall d \in D : \tilde{\phi}(i(d)) \sim_E \psi(i(d)) \Leftrightarrow [\, d \sim_D d' \Rightarrow \tilde{\phi}(d) \sim_E \psi(d')\,]$ since $\forall d \in \tilde{D} : i(j(d)) \sim_D d$. □

Let $\sim$ be the equivalence relation induced by *strip* on the several domains. We verify that $\tilde{\Phi} \in (Conf \to \tilde{\Sigma}^{st}) \dashrightarrow (Conf \to \tilde{\Sigma}^{st})$. Choose $O, O' \in Conf \to \tilde{\Sigma}^{st}$ s.t. $O \sim O'$ and pick $C \in Conf$. If $C = [\,\varepsilon,\sigma\,]$, then $\tilde{\Phi}OC = \sigma \sim \sigma = \tilde{\Phi}O'C$. If $C \to C'$ then $\tilde{\Phi}OC = \tau.O(C) \sim \tau.O'(C) = \tilde{\Phi}O'C$ since by

assumption $O \sim O'$ and $x \sim y \Rightarrow \tau.x \sim \tau.y$.

Suppose $D \leq_{i,j} \tilde{D}$, $E \leq_{k,l} \tilde{E}$ and $F \leq_{m,n} \tilde{F}$. Say $E \to F \leq_{K,L} \tilde{E} \to \tilde{F}$ where $K = \lambda\psi.m \circ \psi \circ l$ and $L = \lambda\psi.n \circ \psi \circ k$. Then $D \to E \to F \leq_{I,J} \tilde{D} \to \tilde{E} \to \tilde{F}$ where $I\phi\tilde{d}\tilde{e} = L(\phi(j\tilde{d}))\tilde{e} = m(\phi(j\tilde{d})(k\tilde{e}))$ and $J\tilde{\phi}de = K(\tilde{\phi}(id))e = n(\tilde{\phi}(id)(le))$. Slightly more general we have, if $D_\alpha \leq_{i_\alpha j_\alpha} \tilde{D}_\alpha$ for $\alpha \in \{1,..,n\}$ and $E \leq_{k,l} \tilde{E}$ then $D_1 \to .. \to D_n \to E \leq_{I,J} \tilde{D}_1 \to .. \to \tilde{D}_n \to \tilde{E}$ where $I\Phi\tilde{d}_1..\tilde{d}_n = k(\Phi(j_1\tilde{d}_1)..(j_n\tilde{d}_n))$ and $J\tilde{\Phi}d_1..d_n = l(\tilde{\Phi}(i_1d_1)..(i_nd_n))$. So for Φ: $(\tilde{D}_1 \to .. \to \tilde{D}_n \to \tilde{E}) \to (\tilde{D}_1 \to .. \to \tilde{D}_n \to \tilde{E})$ we have $\Phi \in (\tilde{D}_1 \sim> .. \sim> \tilde{D}_n \sim> \tilde{E}) \sim> (\tilde{D}_1 \sim> .. \sim> \tilde{D}_n \sim> \tilde{E}) \Leftrightarrow d_1 \sim_1 d_1', ..., d_n \sim_n d_n' \Rightarrow \Phi d_1..d_n \sim_E \Phi d_1'..d_n'$. We will use this unraveling of the notion of canonicity in section 5.

Finally in this section we arrive at the theorem that relates least fixed points of a transformation in the function space $\tilde{D} \to \tilde{D}$ to the least fixed point of its retract in the function space $D \to D$. This theorem is strongly related to the Fixed Point Transformation Lemma. (See [BMZ], [Me].)

(3.6) THEOREM *Suppose* $D \leq_{i,j} \tilde{D}$. *Let* $\tilde{\phi}: \tilde{D} \to \tilde{D}$ *be continuous and canonical. Put* $J(\tilde{\phi}) = \phi$. *Then* $\phi: D \to D$ *is continuous with* $\mu\phi = j(\mu\tilde{\phi})$.

PROOF Clearly, ϕ is continuous by definition of J. By canonicity of $\tilde{\phi}$ we have $\phi \circ j = j \circ \tilde{\phi}$: $\phi(j(d)) = J(\tilde{\phi})(j(d)) = j(\tilde{\phi}(i(j(d)))) = j(\tilde{\phi}(d))$ since $i(j(d)) \sim_D d$.

By induction on n we derive $j(\tilde{\phi}^n(\perp_{\tilde{D}})) = \phi^n(\perp_D)$. Basis, $n=0$: Directly from the strictness of j. Induction step, $n > 0$: $j(\tilde{\phi}^n(\perp_{\tilde{D}})) = j(\tilde{\phi}(\tilde{\phi}^{n-1}(\perp_{\tilde{D}}))) = \phi(j(\tilde{\phi}^{n-1}(\perp_{\tilde{D}}))) = \phi(\phi^{n-1}(\perp_D)) = \phi^n(\perp_D)$ by the equality $j \circ \tilde{\phi} = \phi \circ j$ and the induction hypothesis. By continuity of j we conclude: $j(\mu\tilde{\phi}) = j(\, lub_n\, \tilde{\phi}^n(\perp_{\tilde{D}})) = lub_n\, j(\tilde{\phi}^n(\perp_{\tilde{D}})) = lub_n\, \phi^n(\perp_D) = \mu\phi$. □

4. Operational Semantics and Denotational Semantics for $\mathcal{B}$

In this section we introduce the abstract backtracking language $\mathcal{B}$. This uniform language was studied also in [BV] for it captures the control flow of PROLOG with cut, the latter being the main interest of the particular paper. (See also [BK], [Vi], [Ba2] for similar uses of intermediate abstracta in deriving sound denotational semantics for logic programming languages.) In the present paper however, we will focus on the residue $\mathcal{B}$ on its own to serve as a case study for our method of comparing operational and denotational semantics.

(4.1) DEFINITION Fix a set of actions *Action* and a set of procedure names *Proc*. We define the set of elementary statements $EStat = \{\, a, \underline{fail}, \,!\,, s_1\, \underline{or}\, s_2, x \mid a \in Action, s_i \in Stat, x \in Proc \,\}$, the set of statements $Stat = \{\, e_1:..:e_r \mid r \in \mathbb{N}, e_i \in EStat \,\}$ and the set of declarations $Decl = \{\, x_1 \leftarrow s_1:..:x_r \leftarrow s_r \mid r \in \mathbb{N}, x_i \in Proc, s_i \in Stat, i \neq j \Rightarrow x_i \neq x_j \,\}$. The backtracking language $\mathcal{B}$ is defined by $\mathcal{B} = \{\, d|s \mid d \in Decl, s \in Stat \,\}$.

So a $\mathcal{B}$ program is a declaration together with a statement. Such a statement is a -possibly empty - list of elementary statements of one of the formats action a, procedure variable x, explicit failure $\underline{fail}$, cut operator ! and alternative composition $s_1\, \underline{or}\, s_2$.

We let a range over *Action*, x over *Proc*, e over *EStat*, s over *Stat* and d over *Decl*. We write $x \leftarrow s \in d$ if $x \leftarrow s = x_i \leftarrow s_i$ (for some i) or if $s = \underline{fail}$ otherwise. (By this convention we do not have free procedure variables in a statement, since every x is declared in d having by default the procedure body $\underline{fail}$.)

(4.2) DEFINITION Fix a set Σ of states. Define the set of generalized statements by $GStat = \{ \langle s_1, D_1 \rangle : .. : \langle s_r, D_r \rangle \mid r \in \mathbb{N}, s_i \in Stat, D_i \in Stack \}$. Let γ denote the empty generalized statement. Define the set of frames by $Frame = \{ [g, \sigma] \mid g \in GStat, \sigma \in \Sigma \}$ and the set of stacks by $Stack = \{ F_1 : .. F_r \mid r \in \mathbb{N}, F_i \in Frame \}$.

Next we describe the operational meaning for $\mathcal{B}$. Consider the program $d|s$ and a state σ. The declaration d induces a transition system (also called d). The meaning $\mathcal{O}[\![d|s]\!]\sigma$ then will be the stream of labels of the computation w.r.t. the transition system d starting from an initial configuration associated with s and σ.

We introduce the collection of Σ-transition systems TS by $TS = Stack \rightarrow_{part} (Stack \cup \Sigma \times Stack)$. For $t \in TS$ we shall write $S \rightarrow_t S'$ if $t(S) = S' \in Stack$ and $S \rightarrow_t^{\sigma} S'$ if $t(S) = (\sigma, S') \in \Sigma \times Stack$. We fix an action interpretation $I : Action \rightarrow \Sigma \rightarrow_{part} \Sigma$, that reflects the effect of the execution of an action on a state. (The language $\mathcal{B}$ gains flexibility if actions are allowed to succeed in one state, while failing in another.)

(4.3) DEFINITION Let $d \in Decl$. d induces a transition system d in TS which is defined as the smallest element of TS (with respect to $\subseteq$) such that

(i) $[\gamma, \sigma] : S \rightarrow_d^{\sigma} S$

(ii) $[\langle \varepsilon, D \rangle : g, \sigma] : S \rightarrow_d [g, \sigma] : S$

(iii) $[\langle a ; s, D \rangle : g, \sigma] : S \rightarrow_d [\langle s, D \rangle : g, \sigma'] : S$ if $\sigma' = I(a)(\sigma)$ exists
$[\langle a ; s, D \rangle : g, \sigma] : S \rightarrow_d S$ otherwise

(iv) $[\langle \underline{fail} ; s, D \rangle : g, \sigma] : S \rightarrow_d S$

(v) $[\langle ! ; s, D \rangle : g, \sigma] : S \rightarrow_d [\langle s, D \rangle : g, \sigma] : D$

(vi) $[\langle x' ; s, D \rangle : g, \sigma] : S \rightarrow_d [\langle s', S \rangle : \langle s, D \rangle : g, \sigma] : S$ if $x' \leftarrow s' \in d$

(vii) $[\langle (s_1 \ \underline{or} \ s_2) ; s, D \rangle : g, \sigma] : S \rightarrow_d F_1 : F_2 : S$ where $F_i = [\langle s_i ; s, D \rangle : g, \sigma]$ $(i = 1, 2)$

A stack $S \in Stack$ is a stack of alternatives. Each alternative, i.e. each frame, can be thought of as holding a (partial) elaboration of an initial statement-state pair, also referred to as the original goal. The top frame on the stack is the alternative to be tried first.

If the top frame F holds no proper statements, i.e. $F = [\gamma, \sigma]$, the state σ is outputted on the transition, since the initial goal has been solved yielding σ, and the computation continues with the alternatives embodied by the remainder of the stack. (For we want to deliver all the answers for the initial goal.) If the top frame does contain a proper statement, say $F = [\langle s, D \rangle : g, \sigma]$, an internal transition is made, that depends on the structure of s. The empty component $\langle \varepsilon, D \rangle$ is just skipped.

In case of $a ; s$ the action interpretation I is consulted for the result of action a in state σ. If a transforms σ successfully into a new state σ', the state of the frame F is changed accordingly and the

computation continues with the statement s in F. If a can not be executed successfully in state σ, i.e. $Ia\sigma$ is not defined, this will be a failure for the whole frame F: the alternative is pushed of the stack and the computation continues with the alternatives left on the failure stack S. A explicit *fail* is handled similarly.

A cut can always be executed with success. But, there is a side effect. To implement this side-effect we make use of the cut information represented by the dump stack associated with a statement. This dump stack contains the alternatives that were open at the moment the statement was introduced. Executing a cut means restoring this alternatives and amounts to removal of the alternatives that were created after this (occurrence of) ! was introduced. So in the right-hand side the failure stack S will be replaced by the dump stack D.

In case of a procedure call we apply body replacement. Thus we introduce a new statement, viz. s' in the top frame. Since S consists of the alternatives that are open at this creation time of s' we attach to s' the stack S as its dump stack. In case of an alternative composition s_1 *or* s_2 the to frame splits into two frames. The uppermost corresponding to s_1, the other associated with s_2. So the alternative induced by s_1 will be tried first.

Let Σ^{st} denote the cpo of streams over Σ. We will associate with a declaration d and its induced transition system $\rightarrow_d$ an answer function α_d: $Conf \rightarrow \Sigma^{st}$ that for stacks S yields the concatenation of the σ-labels of the computation starting from C according to the transition system d. We use a higher-order transformation Φ_d for a fixed point definition of α_d.

(4.4) DEFINITION Let $d \in Decl$. Define $\Phi_d \in [(Conf \rightarrow \Sigma^{st}) \rightarrow (Conf \rightarrow \Sigma^{st})]$ by $\Phi_d(\alpha)(E) = \varepsilon$, $\Phi_d(\alpha)(S) = \alpha(S')$ if $S \rightarrow_d S'$, $\Phi_d(\alpha)(S) = \sigma \cdot \alpha(S')$ if $S \rightarrow^{\sigma}_d S'$. The answer function α_d: $Conf \rightarrow \Sigma^{st}$ associated with the Σ-transition system d is defined by $\alpha_d = lfp(\Phi_d)$.

It is straightforward to check that $\bar{\Phi}$ is well-defined, so indeed has a least fixed point. This answer function is used to formulate the operational semantics for $\mathcal{B}$.

(4.5) DEFINITION The operational semantics $\mathcal{O}$: $\mathcal{B} \rightarrow \Sigma \rightarrow \Sigma^{st}$ for the backtracking language $\mathcal{B}$ is defined by $\mathcal{O}(d|s)(\sigma) = \alpha_d([\langle s, E\rangle, \sigma])$ where α_d is the answer function associated with d.

(4.6) DEFINITION

(i) Domains: We define the set of failure continuations $FCont = \Sigma^{st}$, the set of cut continuations $CCont = \Sigma^{st}$, the set of success continuations $SCont = [FCont \rightarrow [CCont \rightarrow \Sigma \rightarrow \Sigma^{st}]]$, the set of meanings $Meaning = Stat \rightarrow [SCont \rightarrow [FCont \rightarrow [CCont \rightarrow \Sigma \rightarrow \Sigma^{st}]]]$. We denote by σ, ϕ, κ, ξ and M typical elements of Σ, $FCont$, $CCont$, $SCont$ and $Meaning$, respectively.

(ii) Functions: The denotational semantics $\mathcal{D}$: $\mathcal{B} \rightarrow \Sigma \rightarrow \Sigma^{st}$ for the backtracking language $\mathcal{B}$ is defined by $\mathcal{D}(d|s)(\sigma) = M_d[\![s]\!] \xi_o \phi_o \kappa_o \sigma$ where $\xi_o = \lambda\phi\kappa\sigma.\sigma \cdot \phi$ and $\phi_o = \kappa_o = \varepsilon$, where M_d is the least fixed point of $\Psi_d \in [Meaning \rightarrow Meaning]$ defined by

$\Psi_d M[\![\varepsilon]\!] \xi\phi\kappa\sigma = \xi\phi\kappa\sigma$

$\Psi_d M[\![a]\!] \xi\phi\kappa\sigma = \xi\phi\kappa\sigma'$ if $\sigma' = I(a)(\sigma)$ exists

$\Psi_d M[\![a]\!] \xi\phi\kappa\sigma = \phi$ otherwise

$\Psi_d M[\![\, \underline{fail} \,]\!] \xi\phi\kappa\sigma = \phi$
$\Psi_d M[\![\ !\ ]\!] \xi\phi\kappa\sigma = \xi\kappa\kappa\sigma$
$\Psi_d M[\![\, s_1 \, \underline{or} \, s_2 \,]\!] \xi\phi\kappa\sigma = M[\![\, s_1 \,]\!] \xi\{M[\![\, s_2 \,]\!] \xi\phi\kappa\sigma\}\kappa\sigma$
$\Psi_d M[\![\, x \,]\!] \xi\phi\kappa\sigma = M[\![\, s \,]\!] \{\lambda\bar{\phi}\bar{\kappa}.\xi\bar{\phi}\kappa\}\phi\phi\sigma \quad \text{if } x \leftarrow s \in d$
$\Psi_d M[\![\, e;s \,]\!] \xi\phi\kappa\sigma = M[\![\, e \,]\!] \{M[\![\, s \,]\!] \xi\}\phi\kappa\sigma$

We leave it to the reader to verify the well-definedness of Ψ but comment briefly on the intuition behind the clauses above.

The transformation is triggered by the statement s. In case of an empty statement we consider the initial goal to be So the success continuation is applied on the particular arguments. In case of a primitive action a that transforms the state σ successfully into the state σ' we also apply the success continuation but now to the new state σ'. If a fails in state σ we deliver the failure continuation ϕ as a denotation. Analogously for the explicit $\underline{fail}$. A cut operator can always be executed successfully but as a side effect the failure continuation is replaced by the cut continuation. For the alternative composition we evaluate the first alternative s_1 according to the meaning M and add the other alternative s_2 on top of the failure continuation. Procedure calls are handled by means of body replacement. The several continuations are changed appropriately. A sequential composition is denoted by the meaning of its first elementary statement while pushing the remainder into the success continuation.

The denotational semantics for $\mathcal{B}$ can be computed given a program $d|s$ from the least fixed point M_d of the transformation Ψ_d using so called standard continuations. Note the format of the standard success continuation $\xi_o = \lambda\phi\kappa\sigma.\sigma\bullet\phi$. This will amount to delivering all remaining alternatives after the first solution is computed.

5. Relating $\mathcal{O}$ and $\mathcal{D}$

In this section we will relate the operational and denotational semantics for $\mathcal{B}$ of the previous section. This will be done similarly to the case of the simple while language of section 2: We extend the defining transformations Φ and Ψ to less abstract transformations $\tilde{\Phi}$ and $\tilde{\Psi}$. Using the result on retracts we infer from the equivalence of $\tilde{\Phi}$ and (an variant of) $\tilde{\Psi}$ the equivalence of $\mathcal{O}$ and $\mathcal{D}$.

(5.1) DEFINITION The function $strip : \tilde{\Sigma}^{st} \to \Sigma^{st}$ is defined by $strip = \mu P$ where $P \in [\,[\, \tilde{\Sigma}^{st} \to \Sigma^{st} \,] \to [\, \tilde{\Sigma}^{st} \to \Sigma^{st} \,]\,]$ is defined by $P\rho\bot = \bot$, $P\rho\varepsilon = \varepsilon$, $P\rho\sigma.x = \sigma.\rho x$, $P\rho\tau.x = \rho x$.

So $strip$ substitutes ε for finite many τ's and $\bot$ for ω many. By continuity of $strip$ we can easily check the distributivity of $strip$ over $\bullet$, i.e. $strip(x \bullet y) = strip(x) \bullet strip(y)$.

(5.2) LEMMA *Σ^{st} is a retract of $\tilde{\Sigma}^{st}$.*

PROOF By continuity of $strip$ and the inclusion mapping $incl: \Sigma^{st} \to \tilde{\Sigma}^{st}$ it suffices to show $strip \circ incl = id_{\Sigma^{st}}$, i.e. $\forall x \in \Sigma^{st}$: $strip(x) = x$. It is straight forward to show by induction on n: (*) $\forall n \in \mathbb{N}\, \forall x \in \Sigma^n \cup \Sigma^n.\bot$: $strip(x) = x$. Now choose $x \in \Sigma^{st}$ arbitrary. Let $\langle x_n \rangle_n$ be a chain in $\Sigma^* \cup \Sigma^*.\bot$ with least upperbound x. Then we have by continuity of $strip$ and by (*): $strip(x) =$

$lub_n\ strip(x_n) = lub_n\ x_n = x$. □

In the remainder of this section we choose all the retractions, based on *strip* and *incl*, denoted by I, J (but also by *strip*), using the construction for function spaces as described after lemma 3.4.

We continue with the extension of the operational semantics. Now for all transitions we will have a label from $\tilde{\Sigma}$. But except for this, definitions 4.3 and 5.3 are the same. So for example, we again make use of the action interpretation $I : Action \rightarrow \Sigma \rightarrow_{part} \Sigma$ to establish the behaviour of an action a in a state σ. Furthermore, let $\tilde{TS}$ denote the collection of $\tilde{\Sigma}$-transition systems $Stack \rightarrow_{part} \tilde{\Sigma} \times Stack$. We use similar conventions as for Σ-transition systems.

(5.3) DEFINITION Let $d \in Decl$. d induces a transition system d in $\tilde{TS}$ which is defined as the smallest element of $\tilde{TS}$ (with respect to $\subseteq$) such that

(i) $[\gamma, \sigma] : S \rightarrow^{\sigma}_{\$} S$

(ii) $[\langle \varepsilon, D \rangle : g, \sigma] : S \rightarrow^{\varepsilon}_{\S} [g, \sigma] : S$

(iii) $[\langle a; s, D \rangle : g, \sigma] : S \rightarrow^{\varepsilon}_{\S} [\langle s, D \rangle : g, \sigma'] : S$ if $\sigma' = I(a)(\sigma)$ exists

$[\langle a; s, D \rangle : g, \sigma] : S \rightarrow^{\varepsilon}_{\S} S$ otherwise

(iv) $[\langle \underline{fail}; s, D \rangle : g, \sigma] : S \rightarrow^{\varepsilon}_{\S} S$

(v) $[\langle !; s, D \rangle : g, \sigma] : S \rightarrow^{\varepsilon}_{\S} [\langle s, D \rangle : g, \sigma] : D$

(vi) $[\langle x'; s, D \rangle : g, \sigma] : S \rightarrow^{\varepsilon}_{\S} [\langle s', S \rangle : \langle s, D \rangle : g, \sigma] : S$ if $x' \leftarrow s' \in d$

(vii) $[\langle (s_1\ \underline{or}\ s_2); s, D \rangle : g, \sigma] : S \rightarrow^{\varepsilon}_{\S} F_1 : F_2 : S$ where $F_i = [\langle s_i; s, D \rangle : g, \sigma]$ $(i = 1, 2)$

We shall associate with a declaration d an answer function $\tilde{\alpha}_d$: $Stack \rightarrow \tilde{\Sigma}^{st}$ that for stacks S yields the concatenation of the all the labels of the computation starting from S according to the transition system d. As before we use a higher-order transformation $\tilde{\Phi}_d$ for a fixed point definition of $\tilde{\alpha}_d$. Note that we presently also demand canonicity for the transformation $\tilde{\Phi}_d$.

(5.4) DEFINITION Let $d \in Decl$. Define $\tilde{\Phi}_d \in [\ (Stack \rightarrow \tilde{\Sigma}^{st}) \rightsquigarrow (Stack \rightarrow \tilde{\Sigma}^{st})\ ]$ by $\tilde{\Phi}_d(\alpha)(E) = \varepsilon$, $\tilde{\Phi}_d(\alpha)(S) = \theta \cdot \alpha(S')$ if $S \rightarrow^{\theta}_{\S} S'$. The answer function $\tilde{\alpha}_d$: $Stack \rightarrow \tilde{\Sigma}^{st}$ associated with the $\tilde{\Sigma}$-transition system d is defined by $\tilde{\alpha}_d = \mu\tilde{\Phi}_d$.

Canonicity of $\tilde{\Phi}_d$ (as is also the case for its continuity) is straightforward to check: Let $\alpha, \alpha' \in Stack \rightarrow \tilde{\Sigma}^{st}$ s.t. $\alpha \sim \alpha'$. To show: $\tilde{\Phi}_d(\alpha) \sim \tilde{\Phi}_d(\alpha')$, i.e. $\forall S \in Stack$: $strip(\tilde{\Phi}_d(\alpha)(S)) = strip(\tilde{\Phi}_d(\alpha')(S))$. Let $S \in Stack$. W.l.o.g. $S \neq E$. Say $S \rightarrow^{\theta}_{\S} S'$. Then we have $strip(\tilde{\Phi}_d(\alpha)(S)) = strip(\theta \cdot \alpha(S')) = strip(\theta) \cdot strip(\alpha(S')) = strip(\theta) \cdot strip(\alpha'(S')) = strip(\theta \cdot \alpha'(S')) = strip(\tilde{\Phi}_d(\alpha')(S))$ since $\alpha(S') \sim \alpha'(S')$ by assumption.

The pleasant property of the new transformation $\tilde{\Phi}$, as was elaborated upon before, is the uniqueness of its least fixed point.

(5.5) LEMMA *For all $d \in Decl$: $\tilde{\Phi}_d$ has a unique fixed point.*

PROOF Let $d \in Decl$. Uniqueness of $\mu\tilde{\Phi}_d$, which exists by continuity of $\tilde{\Phi}_d$, follows from

$\forall S \in Stack$, for this implies maximality of $\mu\tilde{\Phi}_d$: $\tilde{\alpha}_d(S) \in \bar{\Sigma}^* \cup \bar{\Sigma}^\omega$. Let $S \in \mathcal{S} = \{ \bar{S} \in Stack \mid \tilde{\alpha}_d(\bar{S}) \in \bar{\Sigma}^*.\perp \}$ be of minimal length. Then $S \neq E$, so $S \to_\theta^{\S} S'$ for some $\theta \in \bar{\Sigma}$, $S' \in Stack$. But then $\tilde{\alpha}_d(S') \in \bar{\Sigma}^*.\perp$ is of length strictly less than $\tilde{\alpha}_d(S)$. Conclusion: $\mathcal{S}$ is empty, so $\forall S \in Stack$: $\tilde{\alpha}_d(S) \in \bar{\Sigma}^* \cup \bar{\Sigma}^\omega$. □

Next we check that the new answer function $\tilde{\alpha}_d$ derived from $\tilde{\Psi}_d$ equals the old answer function α_d derived from Ψ_d modulo clock ticks τ.

(5.6) LEMMA *For* $d \in Decl$, $strip(\tilde{\alpha}_d) = \alpha_d$.

PROOF Let $d \in Decl$. By theorem 3.6 it suffices to show: $strip(\tilde{\Phi}_d) = \Phi_d$. This is clear, since $\forall \alpha \in Stack \to \Sigma^{st}$, $S \in Stack$: $J(\tilde{\Phi}_d)(\alpha)(S) = strip(\tilde{\Phi}_d(\alpha)(S)) = strip(\varepsilon) = \varepsilon = \Phi_d(\alpha)(S)$ if $S = E$, and $J(\tilde{\Phi}_d)(\alpha)(S) = strip(\tilde{\Phi}_d(\alpha)(S)) = strip(\theta \bullet \alpha(S)) = strip(\theta) \bullet strip(\alpha(S)) = \Phi_d(\alpha)(S)$ if $S \to_\theta^{\S} S'$. □

Next we formulate the extension of the higher order transformation Ψ. Note that we restrict not only to "continuous" continuations but rather to both "continuous and canonical" ones.

(5.7) DEFINITION

(i) Domains: We define the set of failure continuations $FCont^{\sim} = \bar{\Sigma}^{st}$, the set of cut continuations $CCont^{\sim} = \bar{\Sigma}^{st}$, the set of success continuations $SCont^{\sim} = [\, FCont^{\sim} \rightsquigarrow [\, CCont^{\sim} \rightsquigarrow \Sigma \to \bar{\Sigma}^{st}]\,]$, the set of meanings $Meaning = Stat \to [\, SCont^{\sim} \rightsquigarrow [\, FCont^{\sim} \rightsquigarrow [\, CCont^{\sim} \rightsquigarrow \Sigma \rightsquigarrow \bar{\Sigma}^{st}]\,]\,]$. We denote by σ, ϕ, κ, ξ and M typical elements of $\bar{\Sigma}$, $FCont^{\sim}$, $CCont^{\sim}$, $SCont^{\sim}$ and $Meaning^{\sim}$, respectively.

(ii) Functions: Let $d \in Decl$. By $\tilde{M}_d$ we denote the least fixed point of $\tilde{\Psi}_d \in [\, Meaning^{\sim} \rightsquigarrow Meaning^{\sim}]$ defined by

$\tilde{\Psi}_d M[\![\varepsilon]\!] \xi\phi\kappa\sigma = \tau \bullet \xi\phi\kappa\sigma$

$\tilde{\Psi}_d M[\![a]\!] \xi\phi\kappa\sigma = \tau \bullet \xi\phi\kappa\sigma'$ if $\sigma' = I(a)(\sigma)$ exists

$\tilde{\Psi}_d M[\![a]\!] \xi\phi\kappa\sigma = \tau \bullet \phi$ otherwise

$\tilde{\Psi}_d M[\![\underline{fail}]\!] \xi\phi\kappa\sigma = \tau \bullet \phi$

$\tilde{\Psi}_d M[\![\,!\,]\!] \xi\phi\kappa\sigma = \tau \bullet \xi\kappa\kappa\sigma$

$\tilde{\Psi}_d M[\![s_1 \,\underline{or}\, s_2]\!] \xi\phi\kappa\sigma = \tau \bullet M[\![s_1]\!] \xi \{ M[\![s_2]\!] \xi\phi\kappa\sigma \} \kappa\sigma$

$\tilde{\Psi}_d M[\![x]\!] \xi\phi\kappa\sigma = \tau \bullet M[\![s]\!] \{ \lambda\bar{\phi}\bar{\kappa}.\xi\bar{\phi}\kappa \} \phi\phi\sigma$ if $x \leftarrow s \in d$

$\tilde{\Psi}_d M[\![e ; s]\!] \xi\phi\kappa\sigma = M[\![e]\!] \{ M[\![s]\!] \xi \} \phi\kappa\sigma$

Again it is noteworthy that definitions 5.6 and 6.7 are the same except for occurrences of τ.

It is a matter of routine to check $\forall M \in Meaning^{\sim}$: $\tilde{\Psi}_d M \in Stat \to [\, SCont \to [\, FCont \to [\, CCont \to \Sigma \to \bar{\Sigma}^{st}]\,]\,]$ and that moreover $\forall M, M' \in Meaning^{\sim}$ s.t $M \sim M'$, $\forall s \in Stat$, $\forall \xi, \xi' \in SCont^{\sim}$ s.t $\xi \sim \xi'$, $\forall \phi, \phi' \in FCont^{\sim}$ s.t $\phi \sim \phi'$, $\forall \kappa, \kappa' \in CCont^{\sim}$ s.t $\kappa \sim \kappa'$, $\forall \sigma \in \Sigma$: $\tilde{\Psi}_d M s \xi\phi\kappa\sigma \sim \tilde{\Psi}_d M' s \xi'\phi'\kappa'\sigma$. So by 3.5 $\tilde{\Psi}_d$ is well-defined.

(5.8) LEMMA *For all* $d \in Decl$ *we have* $strip(\tilde{M}_d) = M_d$.

PROOF Let $d \in Decl$. By theorem 3.6 it suffices to show $J(\tilde{\Psi}_d) = \Psi_d$, i.e. for $M \in Meaning$, $s \in Stat$, $\xi \in SCont$, $\phi \in FCont$, $\kappa \in CCont$, $\sigma \in \Sigma$ it holds that $J\tilde{\Psi}_d M[\![s]\!] \xi\phi\kappa\sigma = \Psi_d M[\![s]\!] \xi\phi\kappa\sigma$.

This can be done by a straightforward calculation (relying heavily on the remark at the end of section 3) of which we shall exhibit only a typical case where $s = x'$.

Say $x' \leftarrow s' \in d$. $J\tilde{\Phi}_d M[\![x']\!] \xi\phi\kappa\sigma = J(\tilde{\Phi}_d(IM)[\![x']\!] (I\xi)(I\phi)(I\kappa)\sigma = J(\tau \cdot (IM)[\![s']\!] \{ \lambda\bar{\phi}\bar{\kappa}.(I\xi)\bar{\phi}(I\kappa) \}(I\phi)(I\phi)\sigma) = J(\tau \cdot (IM)[\![s']\!] \{ I(\lambda\phi'\kappa'.\xi\phi'\kappa) \}(I\phi)(I\phi)\sigma) = J(\tau) \cdot JI(M[\![s']\!] \{ \lambda\phi'\kappa'.\xi\phi'\kappa \}\phi\phi\sigma) = M[\![s']\!] \{ \lambda\phi'\kappa'.\xi\phi'\kappa \}\phi\phi\sigma = \Phi_d[\![x]\!] \xi\phi\kappa\sigma$. Here we have used $\lambda\bar{\phi}\bar{\kappa}.(I\xi)\bar{\phi}(I\kappa) = \lambda\bar{\phi}\bar{\kappa}.\xi(J\bar{\phi})(J\kappa) = \lambda\bar{\phi}\bar{\kappa}.\xi(J\bar{\phi})\kappa = I(\lambda\phi'\kappa'.\xi\phi'\kappa)$ and $(IM)[\![s]\!] (I\xi)(I\phi)(I\kappa)\sigma = I(M[\![s]\!] (JI\xi)(JI\phi)(JI\kappa)\sigma = I(M[\![s]\!] \xi\phi\kappa\sigma)$. □

The last step towards the equivalence theorem below is the formulation of the intermediate function $\bar{\mathcal{F}}$ defined on configurations which extends $\bar{M}_d$.

(5.9) DEFINITION Let $d \in Decl$. The mappings $\bar{\mathcal{F}}_d$: $Conf \rightarrow \bar{\Sigma}^{st}$, $Frame \rightarrow FCont^{-} \rightarrow \bar{\Sigma}^{st}$, $GStat \rightarrow FCont^{-} \rightarrow \Sigma \rightarrow \bar{\Sigma}^{st}$ are defined as follows: $\bar{\mathcal{F}}_d[\![E]\!] = \varepsilon$; $\bar{\mathcal{F}}_d[\![F:S]\!] = \bar{\mathcal{F}}_d[\![F]\!] \{ \bar{\mathcal{F}}_d[\![S]\!] \}$; $\bar{\mathcal{F}}_d[\![[g,\sigma]]\!] \phi = \bar{\mathcal{F}}_d[\![g]\!] \phi\sigma$; $\bar{\mathcal{F}}_d[\![\gamma]\!] \phi\sigma = \sigma \cdot \phi$; $\bar{\mathcal{F}}_d[\![[\langle s,D \rangle :g]\!] \phi\sigma = \bar{M}_d[\![s]\!] \{ \lambda\phi\kappa.\bar{\mathcal{F}}_d[\![g]\!] \phi \}\phi\{ \bar{\mathcal{F}}_d[\![D]\!] \}\sigma$.

We leave it to the reader to check the well-definedness of $\bar{\mathcal{F}}_d$. We will check that $\bar{\mathcal{F}}_d$ is a fixed point of the transformation $\tilde{\Phi}_d$. Therefore by lemma 5.5 we have that $\bar{\mathcal{F}}_d$ and $\tilde{\alpha}_d$ coincide.

(5.10) LEMMA *For $d \in Decl$ we have $\tilde{\Phi}_d(\bar{\mathcal{F}}_d) = \bar{\mathcal{F}}_d$.*

PROOF Let $d \in Decl$. We have to verify $\tilde{\Phi}_d(\bar{\mathcal{F}}_d)[\![S]\!] = \bar{\mathcal{F}}_d[\![S]\!]$ for each stack S. We only treat the case $[\langle x';s,D \rangle :g,\sigma] :S$ leaving the other (similar and easier) cases to the reader.

$\bar{\mathcal{F}}_d[\![[\langle x';s,D \rangle :g,\sigma] :S]\!] = \bar{M}_d[\![x']\!] \{ \bar{M}_d[\![s]\!] \xi \}\{ \bar{\mathcal{F}}_d S \}\{ \bar{\mathcal{F}}_d D \}\sigma = \tau \cdot \bar{M}_d[\![s']\!] \{ \lambda\phi\kappa.\bar{M}_d[\![s]\!] \xi\phi\{ \bar{\mathcal{F}}_d D \} \}\{ \bar{\mathcal{F}}_d S \}\{ \bar{\mathcal{F}}_d S \}\sigma = \tau \cdot \bar{M}_d[\![s']\!] \{ \lambda\phi\kappa.\bar{\mathcal{F}}_d[\![[\langle s,D \rangle :g]\!] \}\{ \bar{\mathcal{F}}_d S \}\{ \bar{\mathcal{F}}_d S \}\sigma = \tau \cdot \bar{\mathcal{F}}_d[\![[\langle s',S \rangle :\langle s,D \rangle :g] :S]\!] = \tilde{\Phi}_d \bar{\mathcal{F}}_d[\![[\langle x';s,D \rangle :g,\sigma] :S]\!]$ where $\xi = \lambda\phi\kappa.\bar{\mathcal{F}}_d[\![g]\!] \phi$. □

Finally we have arrived in a position in which we are able to compare the operational and denotational semantics for the abstract backtracking language $\mathcal{B}$.

(5.11) THEOREM $\mathcal{O} = \mathcal{D}$

PROOF Let $d|s \in \mathcal{B}$. By uniqueness of the fixed point of $\tilde{\Phi}_d$ and the above lemma we have $\bar{\mathcal{F}}_d = \tilde{\alpha}_d$. So it follows that $\tilde{\alpha}_d[\![[\langle s,E \rangle,\sigma]]\!] = \bar{\mathcal{F}}_d[\![[\langle s,E \rangle,\sigma]]\!] = \bar{M}_d[\![s]\!] \{ \lambda\phi\kappa.\bar{\mathcal{F}}_d[\![\gamma]\!] \phi \}\{ \bar{\mathcal{F}}_d E \}\{ \bar{\mathcal{F}}_d E \}\sigma = \bar{M}_d[\![s]\!] \{ \lambda\phi\kappa.\sigma \cdot \phi \}\varepsilon\varepsilon\sigma = \bar{M}_d[\![s]\!] \xi_o \phi_o \kappa_o \sigma$. Finally by the lemmas 5.6 and 5.8 we arrive at $\mathcal{O}[\![d|s]\!] \sigma = \alpha_d[\![[\langle s,E \rangle,\sigma]]\!] = strip(\tilde{\alpha}_d[\![[\langle s,E \rangle,\sigma]]\!]) = strip(\bar{M}_d[\![s]\!] \xi_o \phi_o \kappa_o \sigma) = M_d[\![s]\!] \xi_o \phi_o \kappa_o \sigma = \mathcal{D}[\![d|s]\!] \sigma$. □

6. References

[Bd]. M. Badinet, "Proving Termination Properties of PROLOG Programs: A Semantic Approach," pp. 336-347 in *Proc. LICS'88*, Edinburgh (1988).

[Ba1]. J.W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice Hall International,

London (1980).

[Ba2]. J.W. de Bakker, "Comparative Semantics for Flow of Control in Logic Programming without Logic," Report CS-R8840, Centre for Mathematics and Computer Science, Amsterdam (1988).

[BBKM]. J.W. de Bakker, J.A. Bergstra, J.W. Klop, and J.-J.Ch Meyer, "Linear Time and Branching Time Semantics for Recursion with Merge," *Theoretical Computer Science* **34**, pp. 135-156 (1984).

[BK]. J.W. de Bakker and J.N. Kok, "Uniform Abstraction, Atomicity and Contractions in the Comparative Semantics of Concurrent Prolog," in *Proc. FGCS'88, Tokyo.* (1988).

[BKMOZ]. J.W. de Bakker, J.N. Kok, J.-J.Ch. Meyer, E.-R. Olderog, and J.I. Zucker, "Contrasting Themes in the Semantics of Imperative Concurrency," pp. 51-121 in *Current Trends in Concurrency: Overviews and Tutorials*, ed. J.W. de Bakker, W.P de Roever & G. Rozenberg, LNCS **224**, Springer (1986).

[BM]. J.W. de Bakker and J.-J.Ch. Meyer, "Metric Semantics for Concurrency," *BIT* **28**, pp. 504-529 (1988).

[BMZ]. J.W. de Bakker, J.-J.Ch Meyer, and J.I Zucker, "On Infinite Computations in Denotational Semantics," *Theoretical Computer Science* **26**, pp. 53-82 (1983).

[BZ]. J.W. de Bakker and J.I. Zucker, "Processes and the Denotational Semantics of Concurrency," *Information and Control* **54**, pp. 70-120 (1982).

[BV]. A. de Bruin and E.P. de Vink, "Continuation Semantics for Prolog with Cut," pp. 178-192 in *Proc. TAPSOFT'89*, ed. J. Díaz & F. Orejas, LNCS **351** (1989).

[DM]. S.K. Debray and P. Mishra, "Denotational and Operational Semantics for Prolog," *Journal of Logic Programming* **5**, pp. 61-91 (1988).

[JM]. N.D. Jones and A. Mycroft, "Stepwise Development of Operational and Denotational Semantics for Prolog," pp. 281-288 in *Proc. Symposium on Logic Programming*, Atlantic City (1984).

[KR]. J.N. Kok and J.J.M.M. Rutten, "Contractions in Comparing Concurrency Semantics," pp. 317-332 in *Proc. ICALP'88*, ed. T. Lepistö & A. Salomaa, LNCS **317**, Springer (1988).

[Me]. J.-J.Ch. Meyer, *Programming Calculi Based on Fixed Point Transformations: Semantics and Applications*, Dissertation, Vrij Universiteit, Amsterdam (1985).

[MV]. J.-J.Ch. Meyer and E.P. de Vink, "Applicatons of Compactness in the Smyth Powerdomain of Streams," *Theoretical computer Science* **57**, pp. 251-282 (1988).

[Pl]. G.D. Plotkin, "The Category of complete Partial Orders: a Tool for Making Meanings," in *Proc. Summer School on Foundations of Artificial Intelligence and Computer Science* , Pisa (1978).

[Vi]. E.P. de Vink, "Comparative Semantics for Prolog with Cut," Report IR-166, Vrije Universiteit, Amsterdam (1988).

INTERSECTION TYPES FOR COMBINATORY LOGIC

Mariangiola Dezani-Ciancaglini,
Dip. Informatica, Corso Svizzera 185, Torino, Italy

Roger Hindley,
Maths. Div., University College, Swansea SA2 8PP, U.K.

Dedicated to J. W. de Bakker in honour of his 25 years of work in semantics.

ABSTRACT.

Two different translations of the usual formulation of intersection types for λ-calculus into combinatory logic are proposed; in the first one the rule (≤) is unchanged, while in the second one the rule (≤) is replaced by three new rules and five axiom-schemes, which seem to be simpler than rule (≤) itself.

INTRODUCTION.

Intersection types were introduced as a generalization of the type discipline of Church and Curry, mainly with the aim of describing the functional behaviour of all solvable λ-terms. The usual →-based type-language for λ-calculus was extended by adding a constant ω as a universal type and a new connective ∧ for the intersection of two types. With suitable axioms and rules to assign types to λ-terms, this gave a system in which (i) the set of types given to a λ-term does not change under β-conversion, and (ii) the sets of normalizing and solvable λ-terms can be characterized very neatly by the types of their members. (CDV[1981] gives an introduction and motivation of ∧ and ω, and BCD[1983] gives a summary of all the most basic syntactic properties of the system.)

Moreover, in the new type-language we can build λ-models (filter models) in which the interpretation of a λ-term coincides with the set af all types that can be assigned to it. Filter models turn out to be a very rich class containing in particular each inverse-limit space, and have been widely used to study properties of D_∞-λ-models; see BCD[1983], CDHL[1983] and CDZ[1987].

More recently, intersection types have been introduced in the programming language Forsythe, which is a descendent of Algol 60, to simplify the structure of types; see R[1988].

Systems of combinators are designed to perform the same tasks as systems of λ-calculus, but without using bound variables. Curry's type discipline turns out to

be significantly simpler in combinatory logic than in λ-calculus. (For an introduction see HS[1986] Chapter 14.)

We propose here two different formulations of intersection types for combinatory logic. They are both essentially just translations of the λ-calculus system presented in BCD[1983], and have all the properties one would expect. However, there is at least one extra complication in combinatory logic. In the case of λ-calculus, the type-assignment rule (≤) is well known to be replaceable by the simpler rule (η) (§1 below). But in combinatory logic some more care must be taken in choosing a rule to replace rule (≤), and we do not know whether the second system we present below is the simplest possible (see §4).

For background λ-calculus, combinatory logic and type-theory, HS[1986] will be used as a basic reference.

1. INTERSECTION TYPES FOR λ-CALCULUS.

We introduce the intersection type-assigment system following BCD[1983], H[1982] and H[1988].

1.1 DEFINITION. (i) The set T of *intersection types* is inductively defined by:

$\phi_0, \phi_1, \ldots \in T$ (type-variables)

$\omega \in T$ (type-constant)

$\sigma, \tau \in T \Rightarrow (\sigma \to \tau) \in T, \ (\sigma \wedge \tau) \in T.$

(ii) A (*type-assignment*) *statement* is of the form M:σ with $\sigma \in T$ and M a λ-term, called its *subject*. A *basis* B is a set of statements with only distinct variables as subjects. If x does not occur in B, then "B, x:σ" denotes $B \cup \{x{:}\sigma\}$.

On intersection types we define a pre-order relation which formalizes the subset relation and will be used in a type-assignment rule.

1.2 DEFINITION. The *≤ relation* on intersection types is inductively defined by:

$\tau \le \tau$, $\quad \tau \le \tau \wedge \tau$,

$\tau \le \omega$, $\quad \sigma \wedge \tau \le \sigma$, $\ \sigma \wedge \tau \le \tau$,

$\omega \le \omega \to \omega$, $\quad (\sigma \to \rho) \wedge (\sigma \to \tau) \le \sigma \to (\rho \wedge \tau)$,

$\sigma \le \rho \le \tau \ \Rightarrow \sigma \le \tau$,

$\sigma \le \sigma', \ \tau \le \tau' \ \Rightarrow \sigma \wedge \tau \le \sigma' \wedge \tau'$,

$\sigma \le \sigma', \ \tau \le \tau' \ \Rightarrow \sigma' \to \tau \le \sigma \to \tau'$.

1.3 DEFINITION. (i) $TA_\lambda(\wedge,\omega,\leq)$ is the type assignment system defined by the following natural-deduction rules and axioms.

Axioms (ω): $M{:}\omega$ (one axiom for each λ-term M).

Rules:

$$(\rightarrow I)\ \frac{\begin{matrix}[x{:}\sigma] \\ \vdots \\ M{:}\tau\end{matrix}}{\lambda x.M{:}\sigma\rightarrow\tau}\ (*) \qquad (\rightarrow E)\ \frac{M{:}\sigma\rightarrow\tau \quad N{:}\sigma}{MN{:}\tau}$$

$$(\wedge I)\ \frac{M{:}\sigma \quad M{:}\tau}{M{:}\sigma\wedge\tau} \qquad (\wedge E)\ \frac{M{:}\sigma\wedge\tau}{M{:}\sigma} \quad \frac{M{:}\sigma\wedge\tau}{M{:}\tau}$$

$$(\leq)\ \frac{M{:}\sigma \quad \sigma\leq\tau}{M{:}\tau}$$

(*) if x is not free in assumptions above $M{:}\tau$, other than $x{:}\sigma$.

(ii) We write $B \vdash_\lambda M{:}\sigma$ if $M{:}\sigma$ is derivable from the basis B in this system.

The main syntactic property of this type system is the following theorem of invariance under β-equality and η-reduction. (For a proof see CDV[1981] Lemma 1 and Theorem 1, or H[1982] §5.)

1.5 THEOREM. (i) $TA_\lambda(\wedge,\omega,\leq)$ *is invariant under* β*-equality; that is, if* $M =_\beta N$ *and* $B \vdash_\lambda M{:}\sigma$, *then* $B \vdash_\lambda N{:}\sigma$.

(ii) $TA_\lambda(\wedge,\omega,\leq)$ *is invariant under* η*-reduction; that is, if* $z \notin FV(M)$ *and* z *does not occur in* B, *and* $B, z{:}\sigma \vdash_\lambda Mz{:}\tau$, *then* $B \vdash_\lambda M{:}(\sigma\rightarrow\tau)$.

The invariance under η-reduction allows a replacement of rule ($\leq$) which preserves type assignment, as follows.

1.6 DEFINITION. (i) Let $TA_\lambda(\wedge,\omega,\eta)$ be the type-assigment system obtained from $TA_\lambda(\wedge,\omega,\leq)$ by replacing rule $(\leq)$ by

$$(\eta)\quad \frac{(\lambda x.Mx):\sigma}{M:\sigma\ .}\ \text{(if x is not free in M)}$$

(ii) Let $B \vdash_{\lambda\eta} M:\sigma$ denote derivability in the resulting system.

1.7 THEOREM. $TA_\lambda(\wedge,\omega,\leq)$ *and* $TA_\lambda(\wedge,\omega,\eta)$ *are equivalent; that is,*

$$B \vdash_\lambda M:\sigma \iff B \vdash_{\lambda\eta} M:\sigma.$$

This equivalence can be proved directly fairly easily, or by using BCD[1983] (in particular Lemma 4.2, Remark 2.10, and the remark just before 4.3).

2. CORRESPONDENCE BETWEEN λ AND CL.

The reader is assumed to know at least the basic definitions of combinatory logic (see Chapter 2 of HS[1986]). The atomic combinators are assumed here to be **S**, **K**, **I**.

2.1 DEFINITION (*Abstraction in Combinatory Logic*).

(i) A *functional (fnl)* term is any of **S**, **S**X, **S**XY, **K**, **K**X, **I** (for any X,Y).

(ii) We present four alternative definitions for $\lambda^*x.X$. (The second one has been discussed in HS[1986] §§9.34-35, and the other three are common in the literature. Note that the definition of λ^β uses λ^η.)

λ^η: (a) $\lambda^\eta x.Y \equiv \mathbf{K}Y$ if $x \notin FV(Y)$,
(b) $\lambda^\eta x.x \equiv \mathbf{I}$,
(c) $\lambda^\eta x.Ux \equiv U$ if $x \notin FV(U)$,
(f) $\lambda^\eta x.UV \equiv \mathbf{S}(\lambda^\eta x.U)(\lambda^\eta x.V)$ if (a)-(c) do not apply.

λ^β: (a), (b) as above,
(c_β) $\lambda^\beta x.Ux \equiv U$ if $x \notin FV(U)$ and U is fnl,
(f_β) $\lambda^\beta x.UV \equiv \mathbf{S}(\lambda^\eta x.U)(\lambda^\eta x.V)$ if (a)-(c_β) do not apply.

λ^{abf}: (a), (b) as above, and (f) used when (a) and (b) do not apply.

λ^{fab} : (f) $\lambda^{fab}x.UV \equiv \mathbf{S}(\lambda^{fab}x.U)(\lambda^{fab}x.V)$,
(a) $\lambda^{fab}x.y \equiv \mathbf{K}y$ if y is an atom distinct from x,
(b) $\lambda^{fab}x.x \equiv \mathbf{I}$.

2.2 DEFINITION (*H-transformations*). Each abstraction determines an H-mapping from λ-calculus to combinatory logic: $(\lambda x.M)_H \equiv \lambda^* x.(M_H)$. (Details are in HS[1986] Chapter 9.) We call these mappings H_β, H_η, H_{abf}, H_{fab}.

Let X_λ denote the λ-term associated in the standard way with the CL-term X, and let $=_{c\beta}$ denote combinatory β-equality (i.e. $X =_{c\beta} Y \Leftrightarrow X_\lambda =_\beta Y_\lambda$).

2.3 LEMMA. (i) *For all CL-terms* X:

$X_{\lambda H_\eta} \equiv X$, *in particular* $\mathbf{S}_{\lambda H_\eta} \equiv \mathbf{S}$;

$X_{\lambda H_\beta} \equiv X$, *in particular* $\mathbf{S}_{\lambda H_\beta} \equiv \mathbf{S}$;

$X_{\lambda H_{abf}} =_{c\beta} X$ *and* $\mathbf{S}_{\lambda H_{abf}} \not\equiv \mathbf{S}$;

$X_{\lambda H_{fab}} =_{c\beta} X$ *and* $\mathbf{S}_{\lambda H_{fab}} \not\equiv \mathbf{S}$.

(ii) *For all λ-terms* M *and for* H_β *or* H_{abf} *or* H_{fab} : $M_{H\lambda} =_\beta M$.

The proof for H_{abf} is in HS[1986] §§9.20-28, and the others are similar; see HS[1986] §9.35 for hints on the proof for H_β.

3. INTERSECTION TYPES FOR CL-TERMS.

We introduce now an assignment of intersection types to CL-terms which can be viewed as a translation of $TA_\lambda(\wedge,\omega,\leq)$ into combinatory logic. Its relation to $TA_\lambda(\wedge,\omega,\leq)$ will be precisely stated in Theorem 3.3.

In this section, *type-assignment statements* have form $X{:}\sigma$ where X is a CL-term. *Bases* are sets $\{x_1{:}\sigma_1, x_2{:}\sigma_2, \ldots\}$ with $x_1, x_2, \ldots$ distinct, as usual.

3.1 DEFINITION. (i) $TA_{CL\beta}(\wedge,\omega,\leq)$ is the system whose rules are $(\to E)$, $(\wedge I)$, $(\wedge E)$, $(\leq)$, and whose axiom-schemes are (ω) and

$(\to \mathbf{I})$ $\mathbf{I}: \sigma \to \sigma$,

$(\to \mathbf{K})$ $\mathbf{K}: \sigma \to \tau \to \sigma$,

$(\to \mathbf{S})$ $\mathbf{S}: (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$.

(ii) We write $B \vdash_{CL} X{:}\sigma$ if $X{:}\sigma$ is derivable from the basis B in this system.

3.2 LEMMA. (i) $B, x{:}\sigma \vdash_{CL} x{:}\tau \;\Rightarrow\; \sigma \le \tau$.

(ii) *Let* λ^* *be any of* λ^{η}, λ^{β}, λ^{abf}, λ^{fab}. *Then*

$$B, x{:}\sigma \vdash_{CL} Y{:}\tau \;\Rightarrow\; B \vdash_{CL} (\lambda^*x.Y){:}\sigma \to \tau.$$

Proof. (i) By an easy induction on deductions.

(ii) Induction on the deduction of $Y{:}\tau$. We will prove the result for all four λ^*'s at once, and will use the induction hypothesis for λ^{η} in proving the induction step for λ^{β}.

Case 1: $Y{:}\tau$ is $x{:}\sigma$. $\therefore Y \equiv x$, $\therefore \lambda^*x.Y \equiv \mathbf{I}$. But $\mathbf{I}{:}\sigma \to \sigma$ is an axiom.

Case 2: $Y{:}\tau$ is either in B, or is an **S**, **K** or **I** axiom, or an ω-axiom with Y an atom $\not\equiv x$. $\therefore$ Y is an atom and $x \notin FV(Y)$, so $\lambda^*x.Y \equiv \mathbf{K}Y$. Hence, by the axiom $\mathbf{K}{:}\tau \to \sigma \to \tau$ and rule $(\to E)$, $B \vdash_{CL} \mathbf{K}Y{:}\sigma \to \tau$.

Case 3: $Y{:}\tau$ is an ω-axiom. $\therefore \tau \equiv \omega$. Now $(\lambda^*x.Y){:}\omega$ is an ω-axiom. And, since $\sigma \le \omega$, we have $\omega \le \omega \to \omega \le \sigma \to \omega$. Hence $(\lambda^*x.Y){:}\sigma \to \omega$ by rule $(\le)$.

Case 4: The last step in the deduction of $Y{:}\tau$ is $(\le)$ or $(\wedge E)$:

$$\begin{array}{c} B \quad x{:}\sigma \\ \vdots \\ \dfrac{Y{:}\rho}{Y{:}\tau} \;(\rho \le \tau) \end{array}$$

Then $(\sigma \to \rho) \le (\sigma \to \tau)$, so we use the induction hypothesis and rule $(\le)$.

Case 5: Rule $(\wedge I)$:

$$\frac{Y{:}\tau_1 \qquad Y{:}\tau_2}{Y{:}(\tau_1 \wedge \tau_2)} \quad (\tau \equiv \tau_1 \wedge \tau_2).$$

By induction hypothesis, $B \vdash_{CL} (\lambda^*x.Y){:}\sigma \to \tau_i$ for $i = 1,2$. But $(\sigma \to \tau_1) \wedge (\sigma \to \tau_2) \le \sigma \to (\tau_1 \wedge \tau_2)$, so rules $(\wedge I)$ and $(\le)$ give the result.

Case 6: Rule $(\to E)$: Say $Y \equiv UV$, and we have:

$$\frac{\begin{array}{c} B \quad x{:}\sigma \\ \vdots \\ U{:}\rho \to \tau \end{array} \qquad \begin{array}{c} B \quad x{:}\sigma \\ \vdots \\ V{:}\rho \end{array}}{UV{:}\tau}.$$

Subcase 6a: $x \notin FV(UV)$ and $\lambda^*x.(UV) \equiv \mathbf{K}(UV)$. Since $x \notin FV(UV)$, x cannot occur in the given deduction. Hence $B \vdash_{CL} UV{:}\tau$. So by the axiom $\mathbf{K}{:}\tau \to \sigma \to \tau$ and rule $(\to E)$, $B \vdash_{CL} \mathbf{K}(UV){:}\sigma \to \tau$.

Subcase 6c: $V \equiv x$, $x \notin FV(U)$, and $\lambda^*x.(UV) \equiv U$. Since $B, x{:}\sigma \vdash x{:}\rho$, we have $\sigma \leq \rho$ by (i). $\therefore (\rho \to \tau) \leq (\sigma \to \tau)$. But $B \vdash_{CL} U{:}(\rho \to \tau)$ since $x \notin FV(U)$; hence by $(\leq)$, $B \vdash_{CL} U{:}(\sigma \to \tau)$.

Subcase 6f: $\lambda^*x.(UV) \equiv \mathbf{S}(\lambda^{*\prime}x.U)(\lambda^{*\prime}x.V)$ (where $\lambda^{*\prime}$ is λ^* if λ^* is λ^{η} or λ^{abf} or λ^{fab}, but $\lambda^{*\prime}$ is λ^{η} if λ^* is λ^{β}). By induction hypothesis for $\lambda^{*\prime}$, we have $B \vdash_{CL} (\lambda^{*\prime}x.U){:}\sigma \to \rho \to \tau$, $B \vdash_{CL} (\lambda^{*\prime}x.V){:}\sigma \to \rho$. Hence the result, by an $\mathbf{S}$-axiom and $(\to E)$. □

3.3 THEOREM. (i) $B \vdash_{CL} X{:}\tau \iff B \vdash_{\lambda} X_{\lambda}{:}\tau$.

(ii) $B \vdash_{\lambda} M{:}\tau \Rightarrow B \vdash_{CL} M_H{:}\tau$ *for* $H_{\eta}, H_{\beta}, H_{abf}, H_{fab}$.

(iii) *For* H_{β}, H_{abf}, H_{fab}, *we also have the converse of (ii).*

Proof. We prove all parts together. (i) "$\Rightarrow$" is trivial.

(ii): Induction on $\vdash_{\lambda}$. The only difficult case is rule $(\to I)$, which comes by Lemma 3.2.

(iii): Let H be any of H_{β}, H_{abf}, H_{fab}, and let $B \vdash_{CL} M_H{:}\tau$. $\therefore$ by (i)"$\Rightarrow$", $B \vdash_{\lambda} M_{H\lambda}{:}\tau$. But $M_{H\lambda} =_{\beta} M$ by Lemma 2.3(ii). $\therefore$ by Theorem 1.5(i), $B \vdash_{\lambda} M{:}\tau$.

(i)"$\Leftarrow$": Let $B \vdash_{\lambda} X_{\lambda}{:}\tau$. Then $B \vdash_{CL} X_{\lambda H_{\beta}}{:}\tau$ by (ii). $\therefore$ $B \vdash_{CL} X{:}\tau$ because $X_{\lambda H_{\beta}} \equiv X$ by Lemma 2.3(i). □

Note that Theorem 3.3(iii) does not hold for H_{η}. A counter-example is $M \equiv \lambda xy.xy$; we have $M_{H_{\eta}} \equiv \mathbf{I}$ which has type $\phi \to \phi$ in the CL-system (ϕ being a type-variable), but it can be shown that M does not have this type in the λ-system.

The following theorem shows that $TA_{CL\beta}(\wedge,\omega,\leq)$ is invariant under β-equality and η-reduction.

3.4 THEOREM. (i) *If* $B \vdash_{CL} X{:}\tau$ *and* $Y =_{c\beta} X$, *then* $B \vdash_{CL} Y{:}\tau$.

(ii) *If* $B, z{:}\sigma \vdash_{CL} Yz{:}\tau$ *and* $z \notin FV(Y)$ *and* z *is not in* B, *then* $B \vdash_{CL} Y{:}(\sigma \to \tau)$.

Proof. (i): By 3.3(i), (iii) and 1.5(i).

(ii) Induction on the deduction of $Yz:\tau$, as follows.

Axioms: $Yz:\tau$ cannot be an **S**, **K**, **I**-axiom. The only possibility is an ω-axiom, with $\tau \equiv \omega$. But $\omega \leq \omega \to \omega \leq \sigma \to \omega$ (since $\sigma \leq \omega$), so we have

$$\dfrac{\begin{array}{c}(\omega)\text{-ax}\\ Y:\omega\end{array}}{Y:(\sigma\to\omega)}\ (\omega \leq \sigma\to\omega)\ .$$

Rule $(\to E)$: Say we have, for some ρ,

$$\dfrac{Y:\rho\to\tau \qquad z:\rho}{Yz:\tau.}$$

But $z:\rho$ is deduced from $B, z:\sigma$ and z does not occur in B. Hence $\sigma \leq \rho$ by 3.2(i). $\therefore (\rho\to\tau) \leq (\sigma\to\tau)$, so by $Y:(\rho\to\tau)$ and rule $(\leq)$, $B \vdash_{CL} Y:(\sigma\to\tau)$.

Rule $(\leq)$ or $(\wedge E)$: Say we have

$$\dfrac{Yz:\rho}{Yz:\tau}\ (\rho \leq \tau)\ .$$

By induction hypothesis, $B \vdash_{CL} Y:(\sigma\to\rho)$. Hence, by $(\leq)$, $B \vdash_{CL} Y:(\sigma\to\tau)$.

Rule $(\wedge I)$: Say $\tau \equiv (\tau_1\wedge\tau_2)$ and we have

$$\dfrac{Yz:\tau_1 \qquad Yz:\tau_2}{Yz:(\tau_1\wedge\tau_2)}\ .$$

By induction hypothesis, $B \vdash_{CL} Y:(\sigma\to\tau_i)$, $i = 1,2$. $\therefore$ by $(\wedge I)$ and $(\leq)$, since $(\sigma\to\tau_1)\wedge(\sigma\to\tau_2) \leq \sigma\to(\tau_1\wedge\tau_2)$, we have $B \vdash_{CL} Y:\sigma\to(\tau_1\wedge\tau_2)$. □

3.5 NOTE. Following H[1982], let us define the set NTS of *Normal Types* to be the set of all types σ such that: either $\sigma \equiv \omega$ or $\sigma \equiv \sigma_1\wedge\ldots\wedge\sigma_n$ with some bracketing and with each σ_i having the form $\sigma_{i,1}\to\ldots\to\sigma_{i,m(i)}\to\phi_i$. Normal types corresponded closely to the types in CDV[1981], which were slightly more restricted than those in BCD[1983] and later papers, including this one. In H[1982] it was proved that the restriction was trivial, in the sense that every deduction $B \vdash_\lambda M:\tau$ could be paralleled by a deduction $B^* \vdash_\lambda M:\tau^*$ containing only normal types, where the map $*$: $T\to$NTS applied to a type gave its "normal form". But in CL the

restriction seems not to be so trivial. For example, in CL there is a problem with the axiom $\mathbf{I}:(\sigma\wedge\tau)\to(\sigma\wedge\tau)$. The type in this is not normal, and the nearest normal type to it is $((\sigma\wedge\tau)\to\sigma)\wedge((\sigma\wedge\tau)\to\tau)$. So if types were restricted to being normal, quite a complicated form of the axiom scheme for **I** would be needed to give a reasonable equivalence to the λ-system. Similarly for **S** and **K**.

4. REPLACING RULE (≤).

In this section we propose an alternative formulation of intersection type-assignment to CL-terms in which rule (≤) has been replaced by something simpler. Let $\mathbf{B} \equiv \mathbf{S}(\mathbf{KS})\mathbf{K}$ and $\mathbf{B'} \equiv \mathbf{SB}(\mathbf{KI})$.

4.1 DEFINITION. (i) $TA_{CL\beta}(\wedge,\omega,\eta)$ is the system for CL-terms whose axiom-schemes are (ω), (→**I**), (→**K**), (→**S**) and

$$(\mathbf{I}_1) \qquad \mathbf{I}:\sigma\to\omega$$

$$(\mathbf{I}_2) \qquad \mathbf{I}:\omega\to(\omega\to\omega)$$

$$(\mathbf{I}_3) \qquad \mathbf{I}:(\sigma_1\wedge\sigma_2)\to\sigma_i \qquad (i = 1,2)$$

$$(\mathbf{I}_4) \qquad \mathbf{I}:((\sigma\to\tau)\wedge(\sigma\to\rho))\to(\sigma\to(\tau\wedge\rho))$$

and whose rules are (→E), (∧I), (∧E) and

$$(\mathbf{I}_5)\ \frac{\mathbf{I}X:\sigma}{X:\sigma} \qquad (\eta_1)\ \frac{\mathbf{BI}:\sigma}{\mathbf{I}:\sigma} \qquad (\eta_2)\ \frac{\mathbf{B'I}:\sigma}{\mathbf{I}:\sigma}$$

(ii) We write $B \vdash_{CL\eta} X:\sigma$ if $X:\sigma$ is derivable from the basis B in this system.

We shall prove that $TA_{CL\beta}(\wedge,\omega,\leq)$ and $TA_{CL\beta}(\wedge,\omega,\eta)$ are equivalent.

4.2 LEMMA. *If* $\sigma \leq \sigma'$, *then* $\vdash_{CL\eta} \mathbf{I}:\sigma\to\sigma'$.

Proof. Induction on the proof of $\sigma \leq \sigma'$. We consider only the non-trivial cases.
Axiom $\sigma \leq \sigma\wedge\sigma$.

$$\dfrac{\dfrac{\overset{(\mathbf{I}_4)\text{-ax}}{\mathbf{I}:((\sigma\to\sigma)\wedge(\sigma\to\sigma))\to\sigma\to(\sigma\wedge\sigma)} \qquad \dfrac{\overset{(\mathbf{I})\text{-ax}}{\mathbf{I}:\sigma\to\sigma} \quad \overset{(\mathbf{I})\text{-ax}}{\mathbf{I}:\sigma\to\sigma}}{\mathbf{I}:(\sigma\to\sigma)\wedge(\sigma\to\sigma)}(\wedge I)}{\mathbf{II}:\sigma\to(\sigma\wedge\sigma)}(\to E)}{\mathbf{I}:\sigma\to(\sigma\wedge\sigma)}(\mathbf{I}_5)\ .$$

<u>Transitivity</u>: Suppose $\mathbf{I}:\sigma\to\tau$ and $\mathbf{I}:\tau\to\rho$. Deduce $\mathbf{I}:\sigma\to\rho$ thus:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{B}:(\tau\to\rho)\to(\sigma\to\tau)\to\sigma\to\rho \qquad \mathbf{I}:\tau\to\rho}{\mathbf{BI}:(\sigma\to\tau)\to\sigma\to\rho}(\to E)}{\mathbf{I}:(\sigma\to\tau)\to\sigma\to\rho}(\eta_1) \qquad \mathbf{I}:\sigma\to\tau}{\mathbf{II}:\sigma\to\rho}(\to E)}{\mathbf{I}:\sigma\to\rho}(\mathbf{I}_5)\ .$$

<u>Replacement in ∧</u>. Assume $\mathbf{I}:\sigma\to\sigma'$ and $\mathbf{I}:\tau\to\tau'$. Deduce $\mathbf{I}:(\sigma\wedge\tau)\to(\sigma'\wedge\tau')$ thus:

$$\dfrac{\dfrac{\overset{(\mathbf{I}_4)\text{-ax}}{\mathbf{I}:((\sigma\wedge\tau)\to\sigma')\wedge((\sigma\wedge\tau)\to\tau')\to(\sigma\wedge\tau)\to(\sigma'\wedge\tau')} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{B}:(\sigma\to\sigma')\to((\sigma\wedge\tau)\to\sigma)\to(\sigma\wedge\tau)\to\sigma' \quad \mathbf{I}:\sigma\to\sigma'}{\mathbf{BI}:((\sigma\wedge\tau)\to\sigma)\to(\sigma\wedge\tau)\to\sigma'}(\to E)}{\mathbf{I}:((\sigma\wedge\tau)\to\sigma)\to(\sigma\wedge\tau)\to\sigma'}(\eta_1) \quad \overset{(\mathbf{I}_3)\text{-ax}}{\mathbf{I}:(\sigma\wedge\tau)\to\sigma}}{\mathbf{II}:(\sigma\wedge\tau)\to\sigma'}(\to E)}{\mathbf{I}:(\sigma\wedge\tau)\to\sigma'}(\mathbf{I}_5) \quad \dfrac{\text{SIMILAR}}{\mathbf{I}:(\sigma\wedge\tau)\to\tau'}}{\mathbf{I}:((\sigma\wedge\tau)\to\sigma')\wedge((\sigma\wedge\tau)\to\tau')}(\wedge I)}{\mathbf{II}:(\sigma\wedge\tau)\to(\sigma'\wedge\tau')}(\to E)}{\mathbf{I}:(\sigma\wedge\tau)\to(\sigma'\wedge\tau')}(\mathbf{I}_5)\ .$$

<u>Replacement in →</u>. Assume $\mathbf{I}:\sigma\to\sigma'$ and $\mathbf{I}:\tau\to\tau'$. Deduce $\mathbf{I}:(\sigma'\to\tau)\to(\sigma\to\tau')$ as follows. In this deduction, let $\xi\equiv(\sigma\to\tau)$, $\eta\equiv(\sigma\to\tau')$, and $\zeta\equiv(\sigma'\to\tau)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{B}:(\xi\to\eta)\to(\zeta\to\xi)\to\zeta\to\eta \qquad \dfrac{\dfrac{\mathbf{B}:(\tau\to\tau')\to(\sigma\to\tau)\to\sigma\to\tau' \quad \mathbf{I}:\tau\to\tau'}{\mathbf{BI}:(\sigma\to\tau)\to\sigma\to\tau'}(\to E)}{\mathbf{I}:\xi\to\eta}(\eta_1)}{\mathbf{BI}:(\zeta\to\xi)\to\zeta\to\eta}(\to E)}{\mathbf{I}:(\zeta\to\xi)\to\zeta\to\eta}(\eta_1) \qquad \dfrac{\dfrac{\mathbf{B}':(\sigma\to\sigma')\to(\sigma'\to\tau)\to\sigma\to\tau \quad \mathbf{I}:\sigma\to\sigma'}{\mathbf{B'I}:(\sigma'\to\tau)\to\sigma\to\tau}(\to E)}{\mathbf{I}:\zeta\to\xi}(\eta_2)}{\mathbf{II}:\zeta\to\eta}(\to E)}{\mathbf{I}:\zeta\to\eta}(\mathbf{I}_5)\ . \qquad \square$$

4.3 THEOREM. $B \vdash_{CL} X:\sigma \iff B \vdash_{CL\eta} X:\sigma$.

Proof. "$\Rightarrow$": The only thing to show is that ($\leq$) is an admissible rule in $TA_{CL\beta}(\wedge.\omega,\eta)$; that is, to show that if $B \vdash_{CL\eta} X:\sigma$ and $\sigma \leq \tau$, then $B \vdash_{CL\eta} X:\tau$. By Lemma 4.2, $\vdash_{CL\eta} \mathsf{I}:\sigma\to\tau$. Then we can deduce

$$\cfrac{\cfrac{\mathsf{I}:\sigma\to\tau \qquad X:\sigma}{\mathsf{I}X:\tau}\,(\to E)}{X:\tau}\,(\mathsf{I}_5)\ .$$

"$\Leftarrow$": Immediate from 3.4(ii). ◻

4.4 NOTE. Rule ($\leq$) can also be replaced by a strengthened I-axiom-scheme saying $\mathsf{I}:\sigma\to\tau$ $(\sigma \leq \tau)$, and an I-rule:

$$\frac{\mathsf{I}X:\sigma}{X:\sigma}\ .$$

Using this axiom-scheme and rule, we get $X:\sigma \vdash X:\tau$ when $\sigma \leq \tau$, as follows:

$$\cfrac{\cfrac{\mathsf{I}:\sigma\to\tau \qquad X:\sigma}{\mathsf{I}X:\tau}\,(\to E)}{X:\tau}\ .$$

Conversely, the axiom and I-rule are easily proved admissible in $TA_{CL\beta}(\wedge,\omega,\leq)$.

REFERENCES.

BCD[1983] Barendregt, H.P., Coppo, M., Dezani-Ciancaglini, M., **A filter lambda model and the completeness of type assignment,** *J. Symbolic Logic* 48, 931-940.

CDHL[1983] Coppo, M., Dezani-Ciancaglini, M., Honsell, F., Longo, G., **Extended type structures and filter lambda models,** in *Logic Colloquium '82*, ed. G. Longo et al., North-Holland Co., 241-262.

CDV[1981] Coppo, M., Dezani-Ciancaglini, M., Venneri, B., **Functional characters of solvable terms,** *Zeit. Math. Logik* 27, 45-58.

CDZ[1987] Coppo, M., Dezani-Ciancaglini, M., Zacchi, M., **Type-theories, normal forms and D_∞-λ-models,** *Information and Computation* 72, 85-116.

H[1982] Hindley, J.R., **The simple semantics for Coppo-Dezani-Sallé types,** *LNCS* 137, Springer-Verlag, 212-226.

H[1988] Hindley, J. R., **Coppo-Dezani-Sallé types in lambda-calculus, an introduction,** MS, Maths. Divn., University College, Swansea SA2 8PP, U.K.

HS[1986] Hindley, J.R., Seldin, J.P., *Introduction to combinators and λ-calculus*, Cambridge University Press.

R[1988] Reynolds, J.C., **Preliminary design of the programming language Forsythe,** Report CMU-CS-88-159, Computer Science Dept., Carnegie-Mellon University, Schenley Park, Pittsburgh , U.S.A.

A CHARACTERIZATION OF THE STATE SPACES OF ELEMENTARY NET SYSTEMS

A. Ehrenfeucht
Department of Computer Science
University of Colorado at Boulder
Boulder, Co 80309
U.S.A.

and

G. Rozenberg
Department of Computer Science
University of Leiden
Niels Bohrweg 1
P.O. Box 9512
2300 RA Leiden, The Netherlands

ABSTRACT

The state space of an elementary net system can be represented as an edge-labeled directed graph. Here we give an exposition of a characterization (from [ER4]) of the class of edge-labeled directed graphs which correspond to the state space representations of elementary net systems.

INTRODUCTION

The theory of *Petri nets* has originated in the famous paper by C.A. Petri ([P]) and since then this theory has provided a spectrum of concepts and tools facilitating the description and the analysis of concurrent systems (see, e.g., [BRR1], [BRR2], and [Re]). At present the theory of Petri nets is a well-accepted theory of concurrency.

One of the important aspects of this theory is the way in which basic aspects of concurrent systems, such as concurrency, non-determinism and confusion, are identified both conceptually and mathematically. This is best seen in the fundamental system model of the theory called *elementary net systems* (see, e.g., [T] and [Ro]).

The notion of the state space of a concurrent system is one of the basic concepts of the theory of concurrent systems. Within the theory of elementary net systems it is formalized through the notion of the case graph. Understanding the notion of a case graph is one of the important aims of the theory of Petri nets. It increases our understanding of the behaviour of an

elementary net system and it forms an important link to the theory of transition systems.

Recently a characterization of state spaces of elementary net systems was obtained in [ER3] and [ER4]. It is based on the theory of *partial 2-structures* which is related to the theory of *2-structures* (see, e.g., [ER1] and [ER2]).

The aim of this paper is to give an exposition of the characterization result directed to those interested in the theory of concurrent systems and in particular in the theory of Petri nets. In this exposition we will ignore many (interesting!) aspects of the theory of partial 2-structures; rather we will focus on these notions that lead to the characterization result. The paper is somewhat informal - it does not contain proofs which can be found in [ER3] and [ER4]. The reader who is intrigued by this paper is advised to study [ER3] and [ER4] to see how central problems concerning the theory of concurrent systems fit well into the framework of partial 2-structures.

PRELIMIARIES

We assume the reader to be familiar with basic notions of the theory of Petri nets, and in particular with the basic theory of elementary net systems (see, e.g., [Ro] and [T]). We assume that elementary net systems (EN systems) we consider satisfy the following conditions: the underlying net of an EN system is pure and simple, and for each event e of an EN system, $^{\bullet}e \neq \emptyset$ and $e^{\bullet} \neq \emptyset$.

We also assume that the reader is familiar with the rudiments of graph theory and in particular with the notion of an edge-labeled graph.

An *edge-labeled graph* g will be specified in the form $g = (V,E,\Delta)$, where V is the set of *nodes*, Δ is the alphabet of *labels*, and $E \subseteq V \times \Delta \times V$ is the set of *labeled edges*. An *initialized edge-labeled graph* is a system $h = (g,v_{in})$ where g is an edge-labeled graph and v_{in} is a distinguished node of g called the *initial node of* h.

We assume that all graphs we deal with are finite.

Finally, for an ordered pair of sets (A,B), the *ordered symmetric difference of* (A,B) is the pair (A-B, B-A); it is denoted by *osd*(A,B).

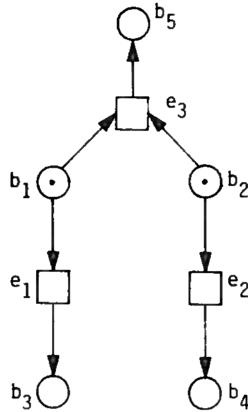
1. ABSTRACT STATE SPACES OF ELEMENTARY NET SYSTEMS

The notion of the *case graph* of an EN system plays an important role in the theory of EN systems - it formalizes the notion of the *state space* of an EN system. It is well known (see, e.g., [RT]) that rather than to consider

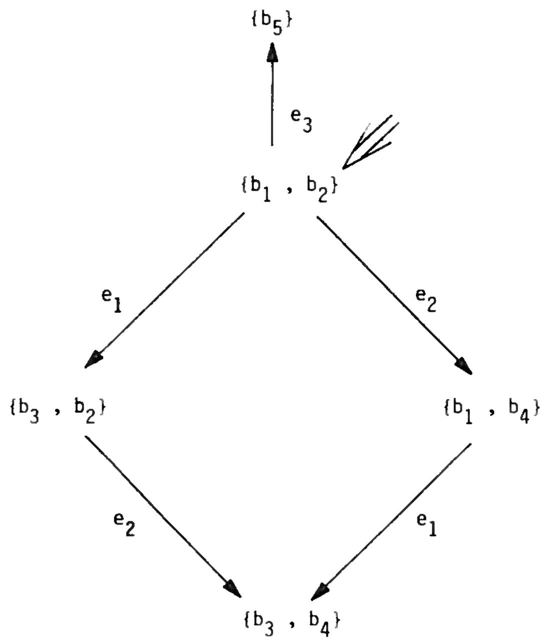
the case graph of an EN system one can consider the *sequential case graph* of an EN system - the case graph can be uniquely recovered from it using the so called "diamond rule".

For an EN system N its sequential case graph will be denoted by SCG(N).

Example 1.1. Consider the following EN system N_0 :



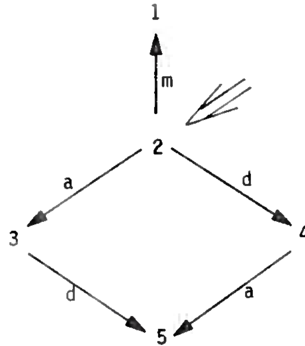
Then SCG(N_0) is as follows:



■

We note that, for an EN system *N*, SCG(*N*) is an initialized edge-labeled graph. If we now take an initialized edge-labeled graph h which is isomorphic with SCG(*N*), then h is an *abstract sequential case graph of N*. Note that nodes of h do not have to be sets and edge-labels may be arbitrary letters; in this way h is an abstract representation of SCG(*N*).

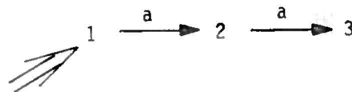
Example 1.2. The following initialized edge-labeled graph is an abstract sequential case graph of the EN system N_0 from Example 1.1:



■

It is easily seen that *not* each initialized edge-labeled graph is an abstract sequential case graph of an EN system.

Example 1.3. The following initialized edge-labeled graph is not an abstract sequential case graph of an EN system:



The reason is that otherwise there must exist an EN system *N*, an event e of *N* (an "interpretation" of the letter a), and a case C_2 of *N* (an "interpretation" of the node 2) such that $e^{\bullet} \subseteq C_2$ and ${}^{\bullet}e \cap C_2 = \emptyset$; a contradiction. ■

The problem discussed in this paper is the problem of characterizing those initialized edge-labeled graphs which are abstract sequential case graphs of EN systems. Once this is achieved we have a characterization of state spaces of EN systems.

2. INITIALIZED LABELED PARTIAL 2-STRUCTURES AND THEIR REGIONS

Initialized labeled partial 2-structures introduced in [ER3] provide a framework for the mathematical theory of state spaces of EN systems. They are defined as follows.

Definition 2.1. An *initialized labeled partial 2-structure*, $i\ell p2s$ for short, is an initialized edge-labeled graph $h = (g,v_{in})$ with $g = (V,E,\Delta)$ such that:

(i) if $(u,a,v) \in E$, then $u \neq v$, and

(ii) if $(u,a,v) \in E$ and $(u,b,v) \in E$, then $a = b$. ∎

Thus an $i\ell p2s$ is an initialized edge-labeled graph without loops and such that between any two nodes there is at most one (labeled) edge. We will use also the notation $h = (V,E,\Delta;v_{in})$ for an $i\ell p2s$ as above.

Clearly, for each EN system *N*, an abstract sequential case graph of *N* is an $i\ell p2s$. On the other hand, for each EN system *N*, SCG(*N*) is an $i\ell p2s$ of a special sort: (1) its nodes are sets, and (2) each edge is labeled by the ordered symmetric difference of sets its connect. The (2) above is based on an additional assumption that we will make in representing SCG(*N*): each event e of *N* is identified with its characteristic pair $({}^{\bullet}e,e^{\bullet})$. This is a natural assumption which we can safely make because we have assumed that we deal with simple sets only.

These special sorts of initialized labeled partial 2-structures will be called initialized labeled partial *set* 2-structures.

Definition 2.2. An *initialized labeled partial set 2-structure*, $i\ell ps2s$ for short, is an $i\ell p2s$ $h = (V,E,\Delta;v_{in})$ such that:

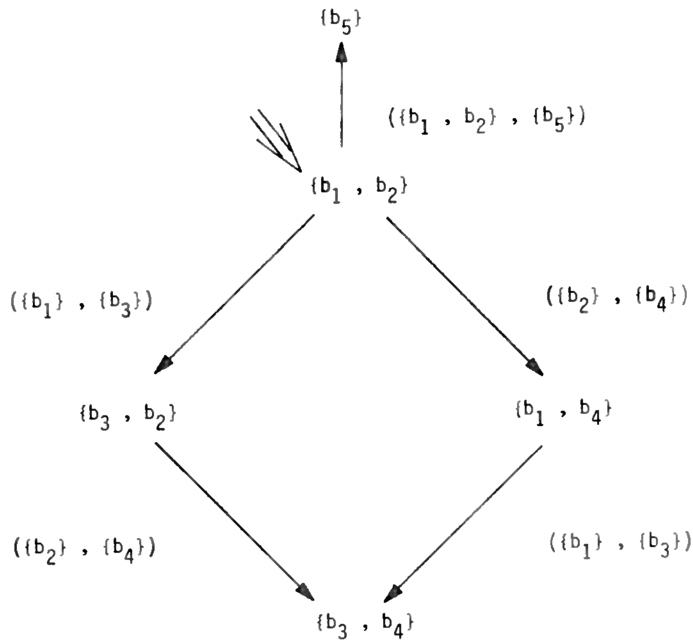
(1) each $x \in V$ is a finite set,

(2) all elements of Δ are of the form (A,B) where A,B are finite sets such $A \cap B = \emptyset$ and $A \cup B \neq \emptyset$, and

(3) for each $(X,A,Y) \in E$, $A = osd(X,Y)$. ∎

In order to avoid often repeating the long acronyms $i\ell p2s$ and $i\ell ps2s$, in the sequel of this paper we will use instead the acronyms p2s and ps2s, respectively.

Example 2.1. According to our convention of identifying an event with its characteristic pair, for the EN system N_0 from Example 1.1, SCG(N_0) is as follows:



Hence it is a ps2s. ■

The notion of an isomorphism between initialized labeled partial 2-structures is as follows.

Definition 2.3. Let $h_1 = (V_1 , E_1 , \Delta_1 ; v^1_{in})$ and $h_2 = (V_2 , E_2 , \Delta_2 ; v^2_{in})$ be initialized labeled partial 2-structures and let φ be a mapping from V_1 into V_2.

(1) φ is a *morphism from* h_1 *into* h_2, iff $\varphi(v^1_{in}) = v^2_{in}$, and for all (x,a,y), $(u,a,v) \in E_1$ such that $\varphi(x) \neq \varphi(y)$ and $\varphi(u) \neq \varphi(v)$, there exists $A \in \Delta_2$ such that $(\varphi(x), A, \varphi(y))$, $(\varphi(u), A, \varphi(v)) \in E_2$.

(2) φ is an *isomorphism from* h_1 *onto* h_2, iff φ is a bijective morphism and φ^{-1} is a morphism. ■

We say that h_1, h_2 are *isomorphic* iff there exists an isomorphism from h_1 onto h_2.

Hence given an EN system N, an initialized edge-labeled graph h is an abstract sequential case graph of N iff h is isomorphic with SCG(N).

The basic technical notion of the theory of (initialized) labeled partial 2-structures is the notion of a region of a p2s. It is defined as follows.

Definition 2.4. Let $h = (V,E,\Delta;v_{in})$ be a p2s and let $Z \subseteq V$. Z is a *region of* g iff for all (x,a,y), $(u,b,v) \in E$, $a = b$ implies that:

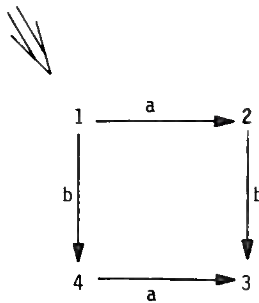
(1) if $x \in Z$ and $y \notin Z$, then $u \in Z$ and $v \notin Z$, and

(2) if $x \notin Z$ and $y \in Z$, then $u \notin Z$ and $v \in Z$. ■

For a subset Z of V and an edge $e = (x,a,y)$ we say that e is *crossing Z* iff either $x \in Z$ and $y \notin Z$ or $x \notin Z$ and $y \in Z$; if the former holds then e is *leaving* Z, and if the latter holds then e is *entering* Z. Hence Z is a region of h iff either all edges of h with the same label are not crossing Z or all edges of h with the same label are crossing Z in the same way, meaning that they are either all leaving Z or all entering Z.

We will use R_h to denote the set of all regions of h and, for an $x \in V$, $R_h(x)$ denotes the set of all regions of h containing x.

Example 2.2. Let h_0 be the following p2s:



Then

R_h = { {1,2,3,4}, {1,2}, {3,4}, {1,4}, {2,3}, ∅ },

$R_h(1)$ = { {1,2,3,4}, {1,2}, {1,4} },

$R_h(2)$ = { {1,2,3,4}, {1,2}, {2,3} },

$R_h(3)$ = { {1,2,3,4}, {3,4}, {2,3} }, and

$R_h(4)$ = { {1,2,3,4}, {3,4}, {1,4}}. ■

3. A CHARACTERIZATION OF THE ABSTRACT SEQUENTIAL CASE GRAPHS OF EN SYSTEMS

In this section we state a characterization result for abstract sequential case graphs of EN systems (proved in [ER3], [ER4]). It is based on the notion of a region, and more explicitely it is based on the following fundamental construction.

Definition 3.1. Let $h = (V,E,\Delta;v_{in})$ be a p2s.

(1) The *regional* h-*mapping*, denoted reg_h, is the mapping φ from V into 2^{R_h} defined by:

for every $x \in V$, $\varphi(x) = R_h(x)$.

(2) The *regional version of* h, denoted *regv*(h), is the ps2s $(V',E',\Delta';v'_{in})$ such that

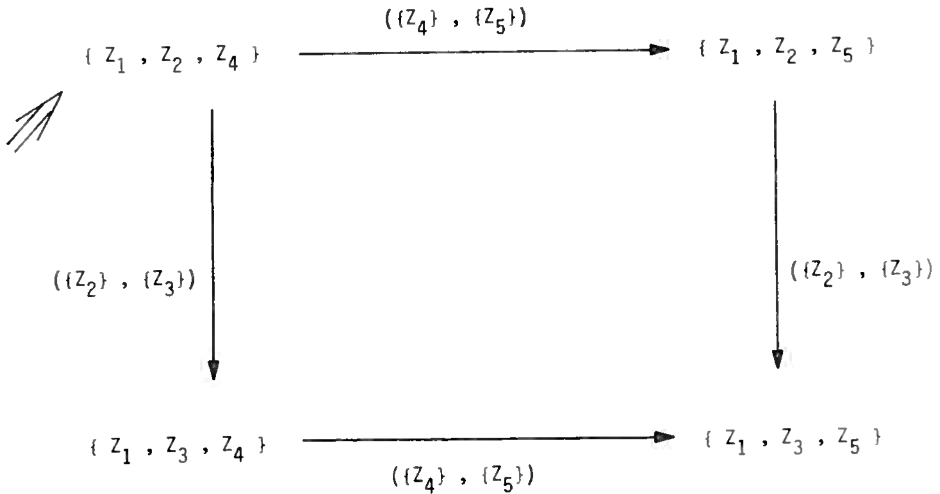
(i) $V' = \{ R_h(x) : x \in V \}$,

(ii) for all $(X,Y) \in V' \times V'$, $(X,osd(X,Y),Y) \in E'$ iff there exists $(x,osd(x,y),y) \in E$ such that $X = R_x$ and $Y = R_y$,

(iii) $\Delta' = \{ osd(X,Y) : (X,osd(X,Y),Y) \in E' \}$, and

(iv) $v'_{in} = \varphi(v_{in})$. ■

Example 3.1. For the p2s h_0 from Example 2.2, $regv(h_0)$ is as follows:



■

To state our main result we need also the following notion.

Definition 3.3. A ps2s $h = (V,E,\Delta;v_{in})$ is *forward closed*, denoted $FC(h)$, iff, for all $(A,B) \in \Delta$ and all $X \in V$ such that $A \subseteq X$ and $B \cap X = \emptyset$, there exists $Y \in V$ such that $(X,(A,B),Y) \in E$. ■

Theorem 3.1. An initialized edge-labeled graph h is an abstract sequential case graph of an EN system iff

(1) h is a p2s,

(2) each node of h is reachable from the initial node of h,

(3) reg_h is an isomorphism, and

(4) $FC(regv(h))$. ■

Example 3.2.

For the p2s h_0 from Example 2.2, reg_{h_0} is an isomorphism of h_0 onto $regv(h_0)$ (see Example 3.1). Clearly each node of h_0 is reachable from the initial node of h_0 , and $FC(regv(h_0))$. Hence by Theorem 3.1, h_0 is an abstract

sequential case graph of an EN system.

On the other hand for the p2s h_1 from Example 1.3, its regional version has one node only and consequently h_1 is not isomorphic with *regv*(h_1). Hence, indeed, h_1 is *not* an abstract sequential case graph of an EN system. ■

REFERENCES

[BRR1] Brauer, W., Reisig, W., and Rozenberg, G., (eds.), *Petri nets : central methods and their properties*, Springer-Verlag, 1987.

[BRR2] Brauer, W., Reisig, W., and Rozenberg, G., (eds.), *Petri nets : applications and relationships to other models of concurrency*, Springer-Verlag, 1987.

[ER1] Ehrenfeucht, A., and Rozenberg, G., Theory of 2-structures, Part I: Clans, morphisms, and basic subclasses, Dept. of Computer Science, University of Leiden, Techn. report No. 88-08, 1988.

[ER2] Ehrenfeucht, A., and Rozenberg, G., Theory of 2-structures, Part II: Representation through labeled tree families, Dept. of Computer Science, University of Leiden, Techn. report No. 88-09, 1988.

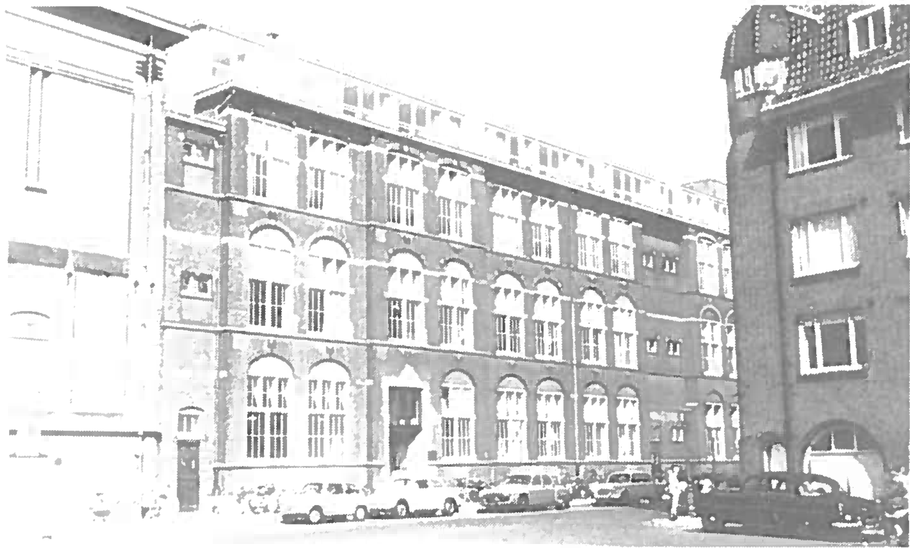
[ER3] Ehrenfeucht, A., and Rozenberg, G., Partial (set) 2-structures, Part I: Basic notions and the representation problem, Dept. of Computer Science, University of Leiden, Techn. report No. 88-10, 1988.

[ER4] Ehrenfeucht, A., and Rozenberg, G., Partial (set) 2-structures, Part II: State spaces of concurrent systems, Dept. of Computer Science, University of Leiden, Techn. report No. 88-11, 1988.

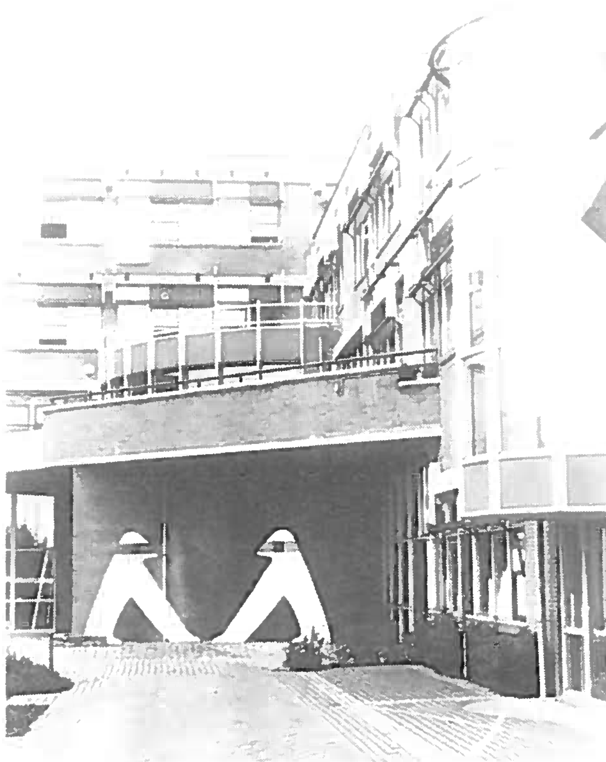
[P] Petri, C.A., Kommunikation mit Automaten, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962 (in German).

[Re] Reisig, W., *Petri nets: an introduction*, Springer-Verlag, 1985.

[RT] Rozenberg, G., and Thiagarajan, P.S., Petri nets: basic notions, structure, behaviour, *in* de Bakker, J.W., de Roever, W.P., and Rozenberg, G., (eds), *Current trends in concurrency*, Springer-Verlag, 585-668, 1986.



Het Mathematisch Centrum in 1964



Het Mathematisch Centrum in 1989

J.W. de Bakker tijdens de promotie van
R.P. van de Riet, 7 februari 1968

Tijdens de IFIP conferentie in St. Andrews, Canada 1977
Foto: E.K. Blum

Tijdens de selectievergadering voor ICALP '80
in het Amsterdamse Oosterpark
vlnr.: J. van Leeuwen, H. Maurer, G. Rozenberg,
A. Salomaa, R. Milner, J.W. de Bakker, M. Paterson

Bij de promotie van J.C. van Vliet, 3 oktober 1979
vlnr.: R.P. van de Riet, J.W. de Bakker, A. van Wijngaarden

De EATCS Council tijdens ICALP '80
in Noordwijkerhout
vlnr.: A. Salomaa, J.W. de Bakker, M. Nivat, C. Böhm,
U. Montanari, R. Milner

De promotie van A. Nijholt, 25 juni 1980
vlnr.: A. Nijholt, J.W. de Bakker

On the use of semantics:

Extending Prolog to a Parallel Object Oriented Language

A. Eliëns
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

This contribution intends to demonstrate how the study of semantics influenced both the implementation and design of a parallel logic programming language, which extends Prolog to a parallel object oriented language.

Classes play a central role in the language. An instance of a class may be asked to evaluate a goal. On creating a process to evaluate a goal the invoking process receives a pointer to that process. The answer substitutions resulting from the evaluation of the goal can, in an asynchronous fashion, be collected by a resumption goal using this pointer. A synchronous communication construct is provided. Communication is blocking, the input side however is allowed to backtrack until an input term is found that unifies with the output term. Non-logical variables, local to an instance of a class, that can hold arbitrary terms as values, may be used to record the state of an instance. Synchronisation between processes referring to an instance of a class can be achieved with an answer statement that allows to postpone the evaluation of a goal until the state of that instance satisfies a particular condition. Instances of classes, and processes evaluating goals, can be allocated explicitly on a processor node, to effect a proper distribution of the computation. Two forms of inheritance are provided, one consists of using clauses of another class, the other consists of copying the non-logical variables of another class into each instance of the class.

- The question is the story itself, and whether or not it means something is not for the story to tell -
Paul Auster, The New York Trilogy.

1. INTRODUCTION

Semantics as other fields of science, knows a gap between theory and practice. I've experienced some of the difficulties in bridging this gap when I made an attempt to apply the techniques of formal semantics to the description of expert systems. Yet, as a tribute to Prof. J.W. de Bakker, who was my guide in this field of science, I would like to sketch how the study of semantics influenced both the implementation and design of a (prototype) parallel Prolog system. This work concerns an effort to augment (standard) Prolog with constructs that allow parallel object oriented processing.

The structure of this paper is as follows: The major part of the paper will be devoted to a description of the language constructs for process-creation, communication and synchronization. Then I will informally describe a continuation semantics derived from [Al86] that underlies the implementation of the sequential Prolog interpreter. The design of a unifying communication construct was influenced

The work in this document was conducted as part of the PRISMA project, a joint effort with Philips Research Eindhoven, partially supported by Esprit 415.

by my studies of the semantics of the synchronous communication construct of Occam [In84]. I will motivate the choice for a particular variant by discussing some of the alternatives. Lastly, I will comment on the current status of the system and return the question back to the field that has inspired me: to provide a formal description of the language extensions proposed.

2. Extending Prolog to a parallel object oriented language

Languages such as Concurrent Prolog and Guarded Horn Clauses have made the choice for implicit parallelism, which means that the parallelism is exploited by the implementation and as such hidden to the programmer. Cf. [Bo89], [Co87]. In contrast, the language extensions that will be proposed provide the programmer with explicit control over the creation (and allocation) of new processes, and with the means to effect synchronisation and communication between processes.

The idea behind these extensions is to lift the constructs for parallelism as present in a parallel object oriented language like POOL [Am87] to a Prolog-like language. A class definition in POOL contains a description of the behavior of an object, a collection of data and methods that operate upon these data. For reasons of protection only methods are allowed access to these data. Each object, that is instance of a class, is associated with a process, in that it may either answer method calls or perform actions on its own. An object may postpone answering certain method calls, dependent upon its state, by means of an answer statement that sets the answer list, the list of methods that are answerable.

Classes, also, play a central role in the language to be proposed. The evaluation of a goal, by an instance of a class, is to some extent analogous to answering a method. A difference with POOL, however, is that in our case the identification of an instance of a class and a process does not hold. A process is created, on explicit demand, for evaluating a goal. The class instance merely provides the functionality needed for evaluating the goal, and possibly some protection and synchronisation.

Examples will be given that show how logic programming and (parallel) object oriented programming may be combined. It turns out, however, that some of the behavior associated with object oriented programming can be achieved without the synchronisation mechanism provided by classes. An Occam-like communication mechanism, extended to deal with unification, seems to suffice for these cases. Cf. [In84], [PN84].

2.1. Classes

Simple classes are just modules, that is collections of clauses, that define the functionality of a process evaluating a goal. A process can be created from (an instance of) a class by asking it to evaluate a goal. Such a process is said to refer to an instance of a class. Each instance may have a multiple of processes referring to it. The evaluation of a goal in itself is sequential, unless other processes are explicitly created.

The true virtue of a class comes into view when non-logical variables are used. Each instance of a class may contain local non-logical variables, that can hold arbitrary terms as values. Mutual exclusion between the evaluation of goals by processes referring to an instance is needed to avoid simultaneous access to these variables. For the purpose of synchronisation, moreover, an answer-statement is provided, to allow an instance to postpone the evaluation of certain goals dependent on its state, as expressed in the values of its non-logical variables. A class thus extends a simple class by the possibility of local non-logical variables, mutual exclusion in the evaluation of goals, and the possibility to postpone the evaluation of goals dependent upon its state. Hence, unless only one instance of a class

is used, these differences necessitate making a copy of a class containing non-logical variables before using it, whereas simple classes need not be copied since no values are shared over different invocations. Only when an answer-statement is encountered, mutual exclusion between goals takes effect. Class constructors, goals having the same name as the class, can be used to initialize an instance with respect to its local variables and its willingness to answer a query.

An example of the declaration of a simple class is given by

```
class list {

member(X,[X|_]).
member(X,[_|T]):- member(X,T).

append([],L,L).
append([H|T],L,[H|R]):- append(T,L,R).

}
```

This class contains no variables or answer-statement, and hence is just a collection of clauses. A class may use other classes by putting them in its use list. Only simple classes may be put in a use list, however, since the variables of the used class cannot be accessed directly.

The extensions to Prolog that allow parallel object oriented programming are first of all a construct for process creation and a means to collect the answer substitutions resulting from the evaluation of a goal. Secondly, a mechanism for synchronous communication will be treated. Next, it will be described how local, non-logical variables and answer statement can be used to govern the behavior of an object, that is its willingness to evaluate goals. Lastly, after dealing with issues of allocation and distribution, a simple form of inheritance will be discussed.

Since the primary intent here is to give an intuition for the mechanisms needed for parallel object oriented logic programming, and to motivate the constructs proposed by examples, the description of the constructs itself will be rather informal.

2.2. Process creation and resumptions

Processes created to evaluate a goal derive their functionality from a class in that the evaluation of the goal takes place by the clauses contained in that class. A goal statement of the form

`Q = new(`*classname*`)!G`

where `G` is an arbitrary term and `Q` a variable, starts up a process derived from the class *classname* to evaluate the goal `G`. The call is *asynchronous* because `Q` is bound to a pointer to the process evaluating the goal. This pointer, the value of `Q`, will for the lack of a better name, be called the *query-status*.

The query-status allows to process the answer substitutions resulting from `G` in the context from which the process for evaluating `G` was split off, by stating the *resumption* goal

`Q?`

which results in the evaluation of a (special) goal that effects the bindings in the current context. Conceptually, this goal amounts to backtracking over the possible solutions until no other solution is available. When backtracking occurs, all answer substitutions provided by `Q?` will be tried before backtracking over any goal preceding `Q?`.

Parallelism is achieved by the fact that the newly created process runs independently of the process that created it, as reflected in the statement sequence

```
Q = new(classname)!G,...,Q?
```

In between the creation of the process and stating the *resumption* goal to collect the answer substitutions, the invoking process can perform whatever action suitable. Backtracking over this sequence will not undo the creation of the process for G. When a goal preceding the sequence however gives rise to a new solution, a new process for evaluating G, possibly differently instantiated, may be created. The occurrence of a cut has effect only locally in the process in which it occurs, in restricting the answer substitutions.

This mechanism of process creation and resumption allows to define and-parallelism in a rather straightforward way, as

```
A&B :- Q = new(this)!A, B, Q?.
```

where `this` refers to the class evaluating the goal `A & B`. Note that, somewhat counter-intuitively, backtracking will be done first over `A` and then over `B`.

The advantage of this approach is that the programmer may restrict the cases where parallel evaluation occurs by imposing extra conditions (cf. [DG84], [HN86]) as in

```
A&B :- ground(A),!, Q = new(this)!A, B, Q?.
A&B :- A, B.
```

where splitting of a new process is allowed only when `A` is ground. Note that to avoid unwanted backtracking over the second solution of `A&B`, the cut in the first clause is necessary.

A typical example of the usage of this kind of parallelism is the following, familiar, quicksort program.

EXAMPLE 1: *quicksort*

```
class sort {
use list.

sort([],[]).
sort([X|T],S):-
  split(X,T,L1,L2),
  ( sort(L1,S1) & sort(L2,S2) ),
  append(S1,[X|S2],S).

split(X,[],[],[]).
split(X,[Y|T],[Y|L1],L2):-
       X > Y,!,
       split(X,T,L1,L2).
split(X,[Y|T],L1,[Y|L2]):-
       split(X,T,L1,L2).

}
```

Each nonempty list is divided into two sublists, one with values less than the values in the other, that are sorted in parallel and then appended.

2.3. Communication and synchronisation

Simple classes, without non-logical variables, allow in combination with a construct for synchronous communication features associated with object oriented programming. Cf. [ST83], [PN84].

Communication between processes can take place by means of input and output statements over channel-like objects, similar as for example in Occam [In84], or Delta Prolog [PN84]. The difference with Occam, however, is that instead of assignment of a value on the input side a unification of the input and output term occurs on both sides. Moreover, whereas Occam requires the number of channels to be fixed in advance, an indefinite number of channels is allowed.

A new channel can be created by the statement

```
C = new(channel)
```

The evaluation of this statement results in binding the variable `C` to a special channel-like object.
An output statement is of the form

```
C!T
```

with `C` referring to a channel, and `T` to an arbitrary term.
An input statement has the form

```
C?T
```

with similar meanings of `C` and `T`. A simple example of a unifying communication is given by

```
C!f(a,X) & C?f(Y,b)
```

which results in the binding of `X` to `b` and of `Y` to `a`.

The communication is *synchronous*, in that the output side waits until there is an input goal such that the input and output term are unifiable, and similarly the input side waits until an output goal occurs. Communication however is *asymmetric* in that the output side waits until a unifiable input term is available, while the input side is allowed to backtrack once an output term is available. Such behavior can be observed with a goal like

```
C!f(X) & ( C?g(a) ; C?f(a) )
```

with ";" the standard Prolog disjunctive operator.

The motivation for allowing backtracking on the input side is illustrated by the following example of a counter.

EXAMPLE 2: *counter*

```
class ctr {

ctr(C):- run(C,0).

run(C,N):-
        C?inc(N),
        N1 = N + 1,
        run(C,N1).
```

```
run(C,N):-
        C?value(N),
        run(C,N).

run(C,N):-
        C?stop.

}
```

A typical example of the use of such a counter is

```
:-
     C = new(channel),
     Q = new(ctr(C)),
     C!inc(N),
     C!stop.
```

The call `Q = new(ctr(C))` initializes the instance of `ctr` to channel `C`, and is a shorthand for `Q = new(ctr)!ctr(C)`. Since both sides of the communication are blocking, either a catchall of the form `C?_` , where `_` is an anonymous variable, must be present on the input side, or an explicit fail command `fail(C)`, that results in failure and backtracking on the output side, while it succeeds on the input side. Once a communication is successful no more backtracking will occur on the input side. When the output side backtracks only a new attempt at communication may be made.

The example above is adapted from [PN84] and was originally given in [ST83] to demonstrate how object oriented programming can be implemented by continuously running processes receiving a stream of messages.

There is an obvious resemblance with the communication construct proposed for Delta Prolog [PN84]. The possibility to create an arbitrary number of channels however allows for setting up a system of processes in a way that is hard to envisage in Delta Prolog.

As an example of a situation where the number of processes and channels between processes can grow indefinitely large I've taken the solution to generating primes as presented in the user-manual for POOL-T [Au85]. The idea, based on the sieve algorithm of Eratosthenes, is to implement each sieve as an independent process connected to a predecessor by means of an input channel and to a successor process by an output channel. The first process generates odd numbers, and sends it to the first sieve. On creation, a sieve waits for the first incoming number, which will be its prime number. For each subsequent number a sieve checks whether it is divisible by its prime. If this is not the case then the number is sent to the next sieve.

EXAMPLE 3: *prime generator*

```
class driver {

driver(I):-
        C = new(channel),
        Q = new(sieve(C)),
        drive(C,I).

drive(C,I):-
         C!I, J = I+2, drive(C,J).
```

```
}

class sieve {

sieve(C):-
        C?I,
        collect(I),
        C1 = new(channel),
        Q = new(sieve(C1)),
        run(I,C,C1).

run(P,Cin,Cout):-
        Cin?I,
        ( I//P != 0 -> Cout!I | true ),
        run(P,Cin,Cout).

collect(I):- write(I).

}
```

The program is started by the goal

```
:- Q = new(driver(3)).
```

The output can be collected by sending each prime with which a new sieve is initialized to a special process. The goal `I // P != 0` is evaluated by simplifying `I // P` to `I` *modulo* `P` followed by a test whether the result is unequal to zero.

2.4. Non-logical variables

The next language extension I wish to discuss concerns the use of non-logical variables local to an instance of a class to store persistent values.

Again, the counter may serve as an example.

EXAMPLE 4: *counter*

```
class ctr {
var num=0.

inc(N):- N = val(num), val(num) = val(num) + 1.
value(N):- N = val(num).

}
```

This example differs from the previous solution in that the object oriented behavior is not achieved by keeping the value of the counter as an argument in a tail-recursive loop, but as an explicit (non-logical) variable that can be updated by assignment. The declaration

```
var num=0.
```

creates such a variable, and initializes it to zero. The variable can be accessed by the expression `val(num)`. The convention is used that when `val(num)` occurs on the left hand side of an equality that the value on the right hand side is stored as the value of the variable `num`.

Communication with a counter of this kind, incrementing and asking for its value, does not take place by means of input and output statements over channels, but by using the parameters of a goal, as illustrated in

```
:-
   C = new(ctr),
   C!inc(N),
   C!value(N1).
```

Since `C` is bound to an instance of the class `ctr`, and not to a channel, `C!G` defined by

```
C!G :- Q = C!G, Q?.
```

must be considered a synchronised version of remote goal evaluation. The goal `C!inc(N)` thus amounts to binding the variable `N` to zero, the value of the local variable `num`, with as a side-effect an increment of this variable by one. This example directly illustrates the analogy between answering a method call and evaluating a goal as hinted at in discussing POOL. Side-effects will not be undone on backtracking.

2.5. Mutual exclusion and answer statements

Typically, a counter is used by only one process. When an instance of a class is referred to by more than one process, mutual exclusion between the evaluation of goals must be provided, in that no two processes are simultaneously granted access to a non-logical variable of that instance. The example of a semaphore, given below, moreover shows also the need of *answer*-statements to postpone the evaluation of a goal.

EXAMPLE 6: *semaphore*

```
class sema {
var num=0.

sema():- run.
sema(N):- val(num) = N, run.

p():- val(num) = val(num) - 1.
v():- val(num) = val(num) + 1.

run:-
        ( val(num) == 0 -> answer(v) ; answer(p,v) ),
        run.
}
```

The constructor for `sema` causes the semaphore to loop over a conditional that tests the value of the non-logical variable `num`. When its value is zero only calls to `v()` will be answered, otherwise both `p()` and `v()` may occur.

An example of its use is given by

```
cs(S,G):- S!p(), G, S!p().

:- S = new(sema(1)), ( cs(S,g1) & cs(S,g2) ).
```

which enforces that the goals `g1` and `g2` will be executed in a non-overlapping fashion. Since the new semaphore is initialized with one, the first process that asks for `p()` must first ask for `v()` before the

other process is granted the evaluation of `p()`.

Operationally, when an answer statement is reached, the evaluation of the current goal is suspended until a goal allowed by the answer list has been evaluated. When the evaluation of a goal not occurring in the answer list is asked for, the process asking for the evaluation is suspended. The process continues when an answer statement occurs that allows it, and moreover, it is first in the queue of (allowed) processes waiting for the evaluation of a goal. The interplay between processes evaluating a goal and the instance of the class to which they commonly refer is rather subtle. It is the instance of the class that decides which processes are to be suspended and which processes may be active in evaluating a goal. Only one process may be active in evaluating a goal. It stops being active either upon termination, or by evaluating an answer statement. In both cases that particular instance may grant the evaluation of goals, as long as they are allowed by the answer list.

2.6. Dining philosophers

Our pièce de résistance is an implementation of the solution to the problem of the *dining philosophers*, as given in [Au85].

Five philosophers must spent their time *thinking* and *eating*. When a philosopher wants to eat he must get hold of two forks. Since only five forks are available, each philosopher must await his turn. To avoid the situation where each philosopher sits waiting with one fork, only four philosophers at a time are allowed in the dining room.

Since a philosopher needs to know no more than his name, the dining room and his proper forks, after creation he may proceed to enter the cycle of thinking, entering the dining room, picking up his forks, eating and leaving the dining room to start thinking, again.

```
class philosopher {
var name.

philosopher(Name,R,Lf,Rf) :-
        val(name) = Name,
        proceed(R,Lf,Rf).

think :- N = val(name), write(thinking(N)).
eat:- N = val(name), write(eating(N)).

proceed(R,Lf,Rf):-
        think,
        R!enter(),
        Lf ! pickup(), Rf!pickup(),
        eat,
        Lf ! putdown(), Rf!putdown(),
        R!exit(),
        proceed(R,Lf,Rf).
}
```

A philosopher is admitted to the dining room when less than four guests are present, otherwise he must wait for one of his collegues to exit.

```
class room {
var occupancy=0.

room():-
        (
        val(occupancy) == 0 -> answer(enter) ;
        val(occupancy) == 4 -> answer(exit) ;
        answer(enter, exit)
        ),
        room().

enter():- val(occupancy) = val(occupancy) - 1.
exit():- val(occupancy) = val(occupancy) + 1.

}
```

Forks can only be picked up and then put down.

```
class fork {

pickup().
putdown().

fork() :-
        answer( pickup ),
        answer( putdown ),
        fork().

}
```

The ceremony is started by assigning the philosophers their proper forks and showing them the dining room. The details of their initiation will, for reasons of space, be omitted.

The example fully demonstrates the synchronisation enforced by answer statements. Such behavior could not be effected by using simple classes and the synchronous communication construct presented earlier. Much worse, the synchronisation capability of classes makes such a communication construct superfluous. My personal preference, however, is to use the communication construct in cases where the sophisticated synchronisation provided by the answer statement is not demanded.

2.7. Distribution and allocation

In the examples given no attention has been paid to the issue of allocating class instances and the actual distribution of computation over the available resources.

When a new instance of a class is created, it can be allocated on a particular processor node, by a statement of the form

`O = new(`*classname*`)@N`

where `N` is the number of a processor node. Also it is possible to split of a process to evaluate a goal on a particular node by the statement `G@N` for `G` a goal, and `N` a node number. Conceptually, the meaning of a goal `G@N` is given by the clause

```
G@N :- O = new(this)@N, Q = O!G, Q?.
```

Since allocation is dependent on assumptions concerning the parallel machine on which the system is implemented these assumptions will be made explicit first.

The parallel processor for which the system is intended, is assumed to have a limited number (less than 100) of processor nodes that are connected with each other by a packet switching network, such that the distance between each node never exceeds a fixed number (3 or 4) of intermediate nodes. For reasons of optimal utilization such machines are considered to support coarse grain parallelism, which means that the ratio of communication and computation must be in favor of the latter.

The programmer knows each processor just by its number, $0 \cdots n-1$, for n processors. However to permit a more refined strategy of allocating processes and resources the language also allows *node expressions* from which a node number can be calculated. Apart from viewing the network as a sequence the programmer can index the nodes as a tree, as in

$$I_0 : I_1 : \cdots : I_n$$

which denotes, with the branching degree (by default) fixed to two, the processor associated to the node in the tree reached by following the path $I_1,...,I_n$ from I_0. The association of processor numbers to nodes of the tree is done by counting the nodes of tree breadth first. For example the expression $0:1:2:1$, giving the path 1,2,1 from 0, results in processor number 9.

As an example of how node expressions can be used to distribute the load of computation over the available processors consider the following variant of the quicksort program presented earlier.

```
class sort {
use list.

sort(L,R):-  sort(0,L,R).

sort(N,[],[]).
sort(N,[X|T],S):-
  split(X,T,L1,L2),
  ( sort(N:1,L1,S1)@N:1 & sort(N:2,L2,S2)@N:2 ),
  append(S1,[X|S2],S).

  . . .

}
```

When splitting off two processes for sorting the sublists, the processes are allocated on the successor nodes of the current node, as long as sufficient processors are available. More refined strategies may be encoded by including tests on the length of the lists.

The processor topology may also be viewed as a matrix with a certain width (say 4 for 16 processors). The programmer can index this matrix by expressions of the form

$$N_1 \# N_2$$

which allows to distribute the load over the available processors by moving, for instance north from $N_1 \# N_2$ to $N_1 \# N_2 + 1$, over the matrix. Such indexing might be useful for the example of grid computation given in [BL86].

2.8. Inheritance

Two forms of inheritance are provided. A class may put another class in its *use* list. The clauses of that class are then used whenever the class itself provides no clauses for a goal. When such clauses do occur the clauses of the other class will not be used for evaluating that goal. When non-logical variables are declared however, it is usually not possible to use clauses from that class, since a reference to these variables has no meaning for the class in which they are used. A simple solution to this problem is to allow a class to copy the non-logical variables of the other class to each instance of itself. Such course of action is taken for all classes in the *isa* list of a class.
For instance, the declaration

```
class b { var num. }

class a { isa b.  }
```

effects that all instances of a have a non-logical variable num.

The notation

```
class a : b { ... }
```

is used as a shorthand for

```
class a {
isa b.
use b.
 ...
}
```

and allows to define a semaphore by declaring

```
class number { var num=0. }

class counter : number {

get(N):- N = val(num).
put(N):- val(num) = N.
inc():- val(num) = val(num) + 1.
dec():- val(num) = val(num) - 1.

}

class semaphore : counter {

semaphore(N) :- put(N), run.

p():- dec().
v():- inc().

run:-
        get(N),
        ( N == 0 -> answer(v); answer(p,v) ),
        run.
}
```

Hence a `semaphore` is a kind of `counter`, which itself is a kind of `number`. Whenever multiple variables with the same name and different initializations occur, the result is undefined.

3. A sequential interpreter for Prolog

The implementation of the interpreter is based on a continuation semantics for Prolog given by [Al86]. The requirement imposed on the implementation language by this semantics is the possibility to treat functions as first class objects. For this reason, and several others, POOL-X was the natural choice for implementing the prototype.

The main idea of the semantics given in [Al86] is that a theory, that is a list of clauses, is compiled prior to a query into a *database* function that delivers a list of answer substitutions. Once a proper definition of composition of these database functions was found the treatment of classes using other classes did not present any serious problems. Although the intent of this section is to demonstrate how to derive an implementation from a formal semantic definition I will neither present a formal (continuation) semantics, nor the (POOL-X) code derived from it; instead I will merely give an outline of the approach followed in implementing the interpreter.

The implementation is term-oriented, in that all objects that may occur in the language, such as instances of classes, clauses etc, are treated as terms. The type T, for terms allows all the types connected with these objects as its underlying types. A distinction is made between a simple term and a compound term which is a term with a comma as a function-symbol. Compound terms are used to make lists of terms. This characterization of a term allows also clauses and theories to be treated as (lists of) terms. A Prolog clause, for instance, is a term with function symbol ":-" and as arguments a single term for the head and a possibly compound or empty term for the body.

Some preliminaries for dealing with the renaming of variables and substitutions must be given. When a new clause is tried to solve a goal, to rename the variables of the clause, an environment function of type $V \rightarrow V$ is needed, where V is the type for variables. Given such a function e the function call $mape(e,t)$ gives as a result the term t with all variables renamed according to e. For substitutions s of type $S = V \rightarrow T$ a similar function $maps$ is used to effect the substitution s on a term.

The trick in setting up a continuation semantics is the following: to define a function that performs a certain task, think of what can be done in one step, formulate what must be done afterwards, and do the step with as a continuation that what must be done to complete the task.

The continuations used for the Prolog interpreter have type $C = N \times N \times S \rightarrow T$. The first (integer) argument is used to keep track of the depth of a derivation, the second argument is used as a count of the number of variables used, and the third argument, a substitution, is needed to effect the bindings that resulted from evaluating a goal thus far.

Before the evaluation of a goal can be handled the clauses in a program must be compiled to a database-function of type

$$D = Q \times T \times C \times N \times N \times S$$

A function call to a database function d has the form $d(q,t,c,m,n,s)$, where q is the query-status that handles the dynamic aspects of the evaluation of a goal such as checking for cuts, communication with other processes and the like, t the actual goal to be evaluated, c the continuation containing what must be done after the goal t is solved, m the depth of the evaluation, n the number of variables in use and s the substitution built up thus far.

For compiling a theory into a database function , an auxiliary continuation type $K = D \rightarrow D$ is

needed. To define the function *compile* of type $T \times D \times K \rightarrow D$, called by *compile*$(th,d,k)$, three cases must be distinguished. When the theory to be compiled is empty the result is just d, when the theory *th* is compound then a continuation, to compile the rest of the theory, must be prepared and the first clause must be compiled with the continuation just prepared. Compiling a single clause results in the creation of a new database function

$$newd := fn\ (q:Q,\ t:T,\ c:C,\ m,n:N,\ s:S):T$$

and involves setting up a new environment to rename the variables in the clause according to n, a new substitution that sets the variables in the clause to undefined, and the creation of a continuation for evaluating the body of the clause. Then the result of the function *newd* is the result of appending the result of unifying the goal t with the head of the clause with as a continuation the evaluation of the body of the clause, to the result of applying the given database function d to the arguments of *newd*. This way all alternative solutions provided by the theory are searched for. The function *compile* then results in applying the continuation k to *newd*, to allow for the compilation of any succeeding clauses.

The initial call to *compile* is, given a list of clauses *th*, *compile*$(th,nilD,nilK)$. Since *nilD* will be the first database function applied for any goal, the result of *nilD* must naturally be *nil*. The continuation *nilK* also must be empty, but must act as an identity on D.

Classes are allowed to import the theory of other classes via their *use* list. A naive approach would consist of combining the theories of each class into one and then to compile the combined theory. However, composition of database functions is easily defined, and allows a seperate compilation of classes. The function $fn\ co(d_1,d_2:D):D$ results in a database function

$$d := fn(q:Q,\ t:T,\ c:C,\ m,n:N,\ s:S):T$$

defined as the result of applying d_1 to the arguments of d when this is not *nil* and the result of applying d_2 to the arguments of d otherwise.

The function *newd* created when compiling a clause contained a call to a function *unify* with as continuation the evaluation of the body of the clause. The function

$$fn\ unify(a,b:T,\ c:C,\ m,n:N,\ s:S):T$$

must, when one of the terms a or b is a variable, create a substitution *news* to update the given substitution. When the variable is still unbound then the contination c is called with the substitution *news*, otherwise the value bound to the variable must be unified with the other term. When only non-variable terms are involved either a test on equality is performed, in case both terms are constants or, in the case of function terms, a unification of the arguments is tried. Both cases, when successful, result in executing the continuation.

The continuation created for the body of the clause contains a recursive call to a *query* function, from which the database function will initially be called. The function *query* is of type similar to a database function, its main purpose however is to deal with compound goals and to keep account of cuts. The function

$$fn\ query(q:Q,\ t:T,\ c:C,\ m,n:N,\ s:S):T$$

results in the continuation $c(m,n,s)$ when the goal is empty. For a compound goal a continuation for evaluating the rest of the goal is created and the database function, stored in the query-status q, is applied to the first term of the goal. For a simple goal the creation of a continuation is omitted.

When a cut occurs as a goal, the query-status is notified of the occurrence of a cut at that particular depth. The database function produced by *compile* includes a test, consulting the query-status q, to prevent backtracking when a cut has occurred.

For a given goal t the function *query* is called as in *query*$(q,t,yes(q,t),0,n,starts)$, where n is the number of variables occurring in t and *starts* the substitution that sets each variable to undefined. The initial continuation is given by applying the function *yes* to q and t. This function first stores the list of variables occurring in t into the query-status q and then creates a continuation that reports, if the evaluation of the goal is successful, the substitution values of these variables to q. This allows to ask the query-status for the answer substitutions resulting from the evaluation of the goal.

For the occasion I've taken the liberty to give a semantics in prose, giving only the gist of the semantics on which this implementation is based. Nevertheless, I hope to have demonstrated the usefulness of semantics from an implementation point of view, to which I may add the encouraging fact that the actual implementation of what is described is of about the same number of lines as the verbiage representing it.

4. Design issues

The prototype parallel Prolog system was developed to serve as a vehicle for investigating primitives by which to extend Prolog for the purpose of parallel (object oriented) processing.

Each of the examples presented, motivated the choice for a particular construct.

The communication construct, involving channels, was directly inspired by my previous experience in developing a semantics for Occam. A notable difference with the construct of Occam however is that, instead of value assignment, a unification between the input term and the output term must occur in both processes that take part in the communication. My first choice was for a symmetric construct that forced both processes to backtrack when the communication failed. The example of a *counter*, nevertheless, indicated that an asymmetric construct, allowing the input side to backtrack until a unifying input statement occurs, is a more natural choice. This solution slightly complicated the communication protocol, since the output side must be notified when the input side succeeds in finding a proper input statement.

As it turns out, communication over channels seems to be no longer necessary, once classes are provided with the capability to store values in non-logical variables and to postpone the evaluation of a goal until a certain condition is satisfied. Nevertheless, from a semantic point of view, these extensions are rather complex since they necessitate to give an account of the interplay between local states of instances of classes and the derivations performed by processes referring to it. In contrast, adding channels seems to be a relatively simple extension to sequential Prolog.

The variety of examples shows that a parallel object oriented Prolog as proposed allows to solve many of the problems found in parallel programming. More examples are needed however to investigate the full potentiality of a language of this kind.

Moreover, a number of questions need to be answered to get a clear insight in the particularities of such a language. Notably the relations between backtracking, communication, and synchronisation by means of answer statements must be studied. Another problem is how the cut operator affects the behavior of a collection of processes. Cf. [Ko88], [dB88].

Having developed the language thus far, it seems a good moment to hand over these questions to the specialists in the field, to clarify the problems involved.

Acknowledgements
I wish to thank Carel van den Berg, who developed some of the tools without which I would never

have undertaken this effort, and who moreover showed an early interest in this project. Also I would like to thank Joost Kok, for suggesting the name POOLOG, and for showing an interest in working at a semantics for this language. Pierre America is thanked for allowing me to be inspired by his language, and for reading a first draft of this manuscript.

REFERENCES

[Al86] L. Allison *A practical introduction to denotational semantics* Cambridge Computer Science Texts 23, 1986

[Am87] *POOL-T: a parallel object oriented language* in: A. Yonezawa and M. Tokoro (eds.) Object oriented concurrent systems MIT Press 1987

[Au85] L. Augusteijn *POOL-T User Manual* Report Esprit project 415 Doc Nr 0104 Philips Research Laboratory Eindhoven

[dB88] J.W. de Bakker *Comparative semantics for flow of control in logic programming without logic* Report CS-R8840, CWI

[BL86] R. Butler, E. Lusk, W. McCune and R. Overbeek *Parallel logic programming for numeric applications* LNCS 225, 1986, pp. 375-388

[Bo89] *Parallelism in Logic Programming* K. De Bosschere Report DG 89-02 Lab. voor Electronica en Meettechniek, Gent, 1989

[Co85] J. Cohen *Describing Prolog by its interpretation and compilation* CACM, Dec 1985

[Co87] J.S. Conery *Parallel execution of logic programs* Kluwer, Boston 1987

[HA84] R. Hasegawa and M. Amamiya *Parallel execution of logic programs based on dataflow concept* Proc. FGCS 1984, ICOT, pp. 507-516

[HN86] M.V. Hermenegildo and R. Nasr *Efficient management of backtracking in and-parallelism* LNCS 225, 1986, pp. 40-55

[DG84] D. DeGroot *Restricted and-parallelism* Proc. FGCS 1984, ICOT, pp. 471-478

[In84] Inmos LtD *The Occam Programming Manual* Prentice Hall International London 1984

[Ko88] J.N. Kok *A compositional semantics for Concurrent Prolog* Proc. STACS, Bordeaux 1988

[PM85] L. Pereira, L. Monteiro, J. Cunha and J. Aparico *Delta Prolog: a distributed backtracking extension with events* LNCS 225, 1986, pp. 69-83

[PN84] L.M. Pereira and R. Nasr *A distributed Logic programming Language* Proc. FGCS 1884, ICOT, pp. 283-291

[Ri88] G.A. Ringwood *Parlog86 and the dining logicians* Comm. ACM Vol. 31 No. 1 pp. 10-25 1988

[ST83] E. Shapiro, A. Takeuchi *Object-oriented programming in Concurrent Prolog* New Generation Computing, Vol. 1, no. 2 1983

A compositional semantics for the Turing machine

Peter van Emde Boas*

Departments of Mathematics and Computer Science, University of Amsterdam,
Nieuwe Achtergracht 166, 1018 WV Amsterdam
Dedicated to J.W. de Bakker at the occasion of his 25-th anniversary at the CWI
March 9, 1989

Abstract

In the current tradition of constructing compositional semantics for programming languages the machine language of the most popular model in structural complexity theory seems to have been overlooked. We show that this gap in the literature can easily be filled. Notwithstanding the fact that the contrary is generally believed, we show that the standard semantics for Turing machines is compositional, provided the right syntax is used for describing their programs. This presents one more example for Janssen's observation that lack of compositionality in well understood semantic situations is frequently caused by syntactic prejudices and not by real semantic problems.

1 Why a semantics for the Turing machine is needed

The study of semantics of programming languages has been an important research area in computer science in general and, in particular at the CWI (formerly the Mathematical Centre) for the past 25 years. The present volume and the seminar organized at the occasion of de Bakkers 25th anniversary at the CWI gives witness to his important contributions to this research area.

I have always considered myself to be a side-liner watching this game of semantics. When I encountered semantical investigations for the first time about 20 years ago, I was amazed by the amount of intellectual effort invested in issues of programming which I believed at that time to be self evident and well understood. It took only a short period to understand that the gap between our intuitive operational understanding of computational processes and their formal representation by mathematical models is huge. The trails which are designed for traversing this gap are frequented by toads and dragons, with pitfalls and swampy pools everywhere around, bringing misfortune to the careless wanderer. I have also learned that our original semantic intuition remains the only compass which will guide us out of the murky waters. Our intuition promises us the existence of the other side of the river and shows us in which direction to move. If we stick to its course we will not be swept away into a bottomless whirlpool of formalisms and symbol manipulation strategies which remain stuck in the translation.

*also CWI-AP6, Amsterdam

My experience of the past two decades also shows that semantical investigations follow fashions which arrive and go away. When I got introduced into the area characterizing recursive functions by least fixed-point semantics was the hot topic. Next we had a long period where models and proof theories for concurrency formed the backbone of the semantics activities. We have seen the algebraic approach to data types, and more recently we have seen a large amount of interest in the study of semantical problems arising in the weird language we call Prolog. The bare fact that the semantics of this language has become a research topic already presents a convincing argument against the claim that a Prolog has an evident semantics defined by its logic clauses.

A semantic topic which I have never encountered in these studies is the semantics of the Turing machine. This device represents (together with the Random access machine and, more recently, the parallel versions of the RAM) the work-horse of computational complexity theory. Being one of the most general accepted formalisms for universal computability it pops up in almost every corner of theoretical and practical computer science. It is also a device which predates the real computers by almost a decade [9], and a device which has hardly changed its fundamental structure during the half century it has been among us.

As far as semantic theory is concerned it seems that complexity theory still lives within the stone age of pure operationalism. In the basic courses and textbooks of introductory computation theory Turing machines are given an operational interpretation. Turing machine programs are only explicitly constructed in very simple examples and the corresponding exercises. In more advanced theory explicit programs are never specified; every algorithm is suggested and made believed to be programmed on Turing machines by force of intimidation. This practice is justified on the basis of the so-called *Inessential use of Church' thesis: Whatever we feel to be computable can be brought within the scope of our formal models.*

Since there are no real Turing machine programs there is no need for proving that these programs achieve what they are intended to do. Since there is no need to prove programs correct there is no specified semantics relative which these programs should meet their specifications. Neither does there exist a specification language aimed at Turing machines.

Faced with this absence of even the bare minimum of a semantic theory for a machine formalism which is generally believed to be of some importance one might ask for the cause to this deplorable state of affairs. The following observations are in order before trying to answer this question:

- There exists a well understood operational semantics for Turing machine computations. This semantics is presented in all textbooks on introduction to computation theory which describe the model. See for example [3].

- It is common knowledge that proving properties of Turing machine programs is an undecidable problem. Starting from the undecidability of the Halting problem it is an elementary exercise to show that virtually every behavioral property of Turing machines is also undecidable. The same holds for all nontrivial extensional properties (properties which are completely determined by the function computed by the Turing machine), as follows from Rice's theorem [8]

- Proving properties of Turing machine programs is not only hard in theory as exem-

plified by the above undecidability results, but even the small instances are highly intractable. This is illustrated for example by the problem of evaluating the initial values of the so-called busy beaver function: the largest number of 1-symbols printed by a k-state Turing machine program over the alphabet $\{0,1\}$) as a function of k. The evaluation of this function for $k = 5$ was the subject of a competition organized at the occasion of the 1983 GI TCS meeting at Dortmund; the winner only could have obtained his result using special purpose hardware [2]

- It is generally believed that Turing machine programs rank among the most structureless programs one encounters in the universe. Every additional instruction may screw up the meaning of a well understood program in such a way that even the cleverest machine code hackers may fail to understand what's going on

Each of the above four observations can be invoked as an excuse for the absence of a well explored semantics theory of the Turing machine. In the line of the first remark one simply can deny that there exists a problem. We know perfectly well what we are talking about. But since the standard semantics is operational it is evident that this belief is unfounded and should be shattered.

The second observation leads to the excuse that it doesn't make sense to design a theory of Turing machines since we know beforehand that all problems will be unsolvable. This is an invalid argument in the light of the observation that researchers in semantics have never hesitated to look at semantical theories where the underlying validity problem is even far harder. The halting problem of Turing automata is only Σ_1^0-complete whereas many programming logics have Π_1^1-complete validity/satisfiability problems.

The third observations illustrates the lack of well designed paradigmatic examples of solved semantical problems in the area of Turing machine programs. This is just a matter of lack of interest of the research community. If nothing else is invented one can always redo the factorial example or rebuild the towers of Hanoi.

The last argument simply illustrates the fundamental discrepancy between the average researcher in semantics and its true vocation. He should investigate the semantics problems of real programmers using real-life programming languages and not be side-tracked by the clean well-structured toy languages designed by the theoreticians for their own benefit. We simply can't wait until humanity has decided that it will believe the prophets and start to program in a well structured manner [1]. As long as we can't cope with real Turing programs we have not yet fulfilled our task.

I believe therefore that after 25 years time has arrived that the semantics community should attack the problem of the Turing machine by taking its semantics serious. As a small contribution towards completion of this program I will in this note discuss the issue of compositionality of the semantics of Turing machines. This clearly represents a necessary preliminary step since it is generally accepted that there exist no alternative for working compositional in semantics [6].

2 The standard semantics of the Turing machine

For the purpose of this note I will use the single-tape Turing automaton in stead of the usual model from complexity theory where one has special input tapes, output tapes, and

work tapes. It is folklore knowledge that all versions of Turing automata are equivalent in the sense that they can simulate each other with polynomial time overhead and constant factor overhead in space [10]. This implies that all models are equivalent from the perspective of structural complexity theory, and this provides us the freedom of selecting a convenient version to start our semantical investigations on.

It is usual to define a Turing machine by means of a sixtuple of sets. However, for the purpose of semantics it is advantageous to start from a more syntactic definition. For this purpose I introduce a pair of infinite alphabets $Q = \{q_0, q_f, q_1, \ldots\}$ and $\Sigma = \{s_0, s_b, s_1, \ldots\}$. The elements of the alphabet Q are called ***states*** and the elements of Σ are called *(tape)symbols*. The state q_0 is called the ***start state*** and the state q_f is called the *accepting state*. The symbol s_b is called the ***blank symbol***. We denote moreover $M = \{L, 0, R\}$.

Definition 1 A *Turing machine program* is a string A in the language described by the regular expression $([Q\Sigma Q\Sigma M])^*$.

Readers familiar with the traditional definition of a Turing machine will certainly recognize that the above definition yields the quintuple programs of nondeterministic Turing machines. The substrings of the form $[Q\Sigma Q\Sigma M]$ in A are called *instructions*. It should be observed that the observed complexity of Turing machine semantics is therefore certainly not due to the complexity of the syntactic structure. The language of Turing programs as defined above is a simple regular language.

The traditional semantics of the Turing machine is defined in terms of computations and configurations. These concepts have syntactic definitions as well.

Definition 2 A *configuration* is a string c in the language described by the regular expression $\$\Sigma^*Q\Sigma\Sigma^*\$$. An *initial configuration* is described by the regular expression $\$q_0\Sigma\Sigma^*\$$. An *accepting configuration* is described by the regular expression $\$\Sigma^*q_f\Sigma\Sigma^*\$$.

It should be noted that the above definition yields a machine-independent concept of configuration in general and initial and final configurations in particular. Our next notion however requires the interaction between a configuration and a Turing program:

Definition 3 A configuration c is *terminal relative the Turing program A* if the unique substring of the form $Q\Sigma$ in c does not occur embedded in a substring of the form $[Q\Sigma$ in A.

It is usual to express this notion by requiring that for the pair $q_i s_j$ occurring in the configuration c there is no instruction $[q_i s_j q_{i'} s_{j'} m]$ in A. As described above it seems that the property of being a terminal configuration is context sensitive, but for fixed program A the terminal (and therefore also the non-terminal) configurations form a regular language.

The next notion which must be introduced is the notion of a transition. It is common to attach context sensitive production rules to instructions in the program. When this has been done configurations no longer need a separate definition since they become nothing but productions in the grammar described by these production rules.

Definition 4 With the program A we associate the following collection $P(A)$ of context sensitive production rules:

- for every instruction $[q_i s_j q_{i'} s_{j'} 0]$ in A we introduce the production $(q_i s_j \mapsto q_{i'} s_{j'})$ in $P(A)$
- for every instruction $[q_i s_j q_{i'} s_{j'} R]$ in A and every symbol q_k which occurs in A we introduce the productions $(q_i s_j s_k \mapsto s_{j'} q_{i'} s_k)$ in $P(A)$; to this we add the production $(q_i s_j \$ \mapsto s_{j'} q_{i'} s_b \$)$ in $P(A)$
- for every instruction $[q_i s_j q_{i'} s_{j'} L]$ in A and every symbol q_k which occurs in A we introduce the productions $(s_k q_i s_j \mapsto q_{i'} s_k s_{j'})$ in $P(A)$; to this we add the production $(\$ q_i s_j \mapsto \$ q_{i'} s_b s_{j'})$ in $P(A)$
- These are the only productions in $P(A)$

Definition 5 A *transition* between two configurations c and c' relative A, denoted $c \overset{A}{\mapsto} c'$ is a one step production from c into c' by the grammar $P(A)$. A *computation* from c to c' of machine A, denoted $c \overset{A}{\mapsto}^* c'$ is a production from c to c' under the grammar $P(A)$. A *full computation* is a computation starting in an initial state and leading to a terminal configuration. The full computation is *accepting* if its last configuration is.

The crucial part of the above definition is the transition between one step transitions to computations. This is nothing but the step between a binary relation on the set of all configurations to its transitive reflexive closure, which is a well known ingredient in semantics. As we will see in the sequel it is precisely this transitive closure from which the problems of understanding the semantics of Turing machine programs originate.

It remains to assign inputs and results to Turing machine computations. This is again easily done by some syntactic definitions.

Definition 6 The *result* of some configuration c is equal to $U(c)$ where the mapping U is some simple gsm mapping, depending on the purpose of the computation intended.

The most common examples of results covered by this definition are:

- existence of a terminal configuration: here the gsm mapping U tests for the occurrence of a state-symbol pair for which no instruction is available in A. This is the result which is relevant for *accepting* a set by halting.
- existence of an accepting configuration: here the gsm mapping U tests for the occurrence of the accepting state (which is assumed not to be active in any instruction in A). This is the result which is relevant for *recognizing* a set by accepting.
- the result of a function evaluation: here the gsm mapping U tests for the occurrence of the accepting state symbol, but at the same time all occurrences of the \$-symbol, the blank symbol s_b and the state-symbols will be erased.

Definition 7 The *initial configuration* for some string x is the configuration $i(x) = \$q_0 x\$$

It is usual to presume that x is nonempty and that the blank symbol doesn't occur in x. In the case of an empty input the initial configuration $\$q_0 s_b \$$ is used.

The above notions suffice for defining the operational semantics of Turing machine programs. In order to assign to a program A some meaning $M(A)$ which is a multi-valued

partial function from strings to results argues as follows: for the input string x the initial configuration $i(x)$ is constructed. Next all full computations $i(x)\overset{A}{\mapsto}^{*} c'$ are obtained and for every resulting terminal configuration c' the result $U(c')$ is computed. Then $M(A)$ is the function which maps x onto the set of results $U(c')$ obtained in this way.

3 Is the semantics of Turing machines compositional?

In the previous section I have given a definition of the traditional operational semantics of Turing machines. But why is this called an operational semantics? It seems that this name is used only because the semantics has been obtained using computation sequences consisting of configurations connected by computation steps, where each computation step is performed by execution of some instruction in the program. This interpretation, however, only refers to the names we have given to our syntactic objects used in the definitions. In fact the whole definition above is given in terms of well known mathematical structures, like regular languages, production rules, grammars, gsm mappings, a transition relation and its reflexive transitive closure.

If we put everything together we can rephrase the entire definition by the formula:

$$M(A) = \lambda x[U((\overset{A}{\mapsto}^{*})(i(x)))] \qquad (i)$$

It should be self evident that the above formula is as mathematical and as denotational as one might ever require; all operators in this formula have a clear mathematical meaning which, in principle, is easily formalized in the language of set theory. If the above semantics therefore is called operational this is not caused by the semantics itself but rather by the way it has been traditionally been looked upon within computer science.

The more relevant question to ask is whether the semantics as described by formula i is compositional or not. Here I refer to the well known Fregean principle of compositionality which is expressed by the assertion:

The meaning of some compound expression is composed from the meanings of its parts

According to the reconstruction of the algebraic content of the compositionality principle which was given by Theo Janssen in his 1983 dissertation [4], we may obtain a compositional semantics according to i provided we can assign both to the language of Turing machine programs and to its semantics the structure of an algebra, and do so in such a way that the above mapping M becomes a homomorphism. The algebra on the semantic structure may be obtained by combining operators from another algebra like set theory into so-called polynomial operators, according to the guideline *polynomial is safe*. At the same time we may exploit all imaginable tricks in restructuring the syntactic structure of the programs in order to tune the syntactic algebra towards the semantic algebra.

Now the following observations can be made.

1. The transformation from instructions in A to production rules in $P(A)$ is of a straightforward permutational nature; the symbols are slightly permuted. This operation is easily expressed by polynomial operators over the algebra of strings and sets

2. The transformation from a production rules to the set of all single step productions performed using this production is a polynomial operator in the algebra of sets of strings

3. The single step production relation $\mapsto$ according to a set of production rules is the union of the relations according to the individual productions (a clear instance of homomorphic behavior where the operator concatenation is mapped onto union)

4. The transition from single step productions to many step productions which represent computations is represented by the transitive reflexive closure operator which belongs to the algebra of sets and strings and relations

5. The final semantics i is obtained by pre- and post-multiplication with simple mappings belonging to the algebra of strings and sets

The above observations show us how to restructure the syntax of our Turing machine programs. The individual 5 steps are reflected if we restructure the grammar describing Turing machine programs as follows:

```
instruction ::= '['state,symbol,state,symbol,move']'    $ step 1
instruction1 ::= instruction                            $ step 2
program1 ::= (instruction1)*                            $ step 3
program2 ::= program1                                   $ step 4
program  ::= program2                                   $ step 5
```

In this restructuring of the grammar we have introduced three unit-rules which express the changes of types for the underlying semantic objects from (set of) rules to subsets of the transition relation, and from the transition relation to its transitive closure and next to its input-result relation. Such lifting rules are a common tool in Montague grammar.

It should be noted at this point that we have obtained a compositional semantics for Turing machine programs without modifying the traditional semantics (widely believed to be an operational semantics) just by reconsidering the semantics and slightly restructuring the syntax. The key idea is that the juxtaposition of instructions should be mapped onto the union of relations, and that this union should be performed before subjecting the transition relation to the transitive closure (as expressed by the lifting rules).

One could make the objection that the second step in the semantics described above, which transforms production rules into single step productions requires knowledge of the complete tape alphabet of the involved Turing machine. There are several solutions to this problem. One can specify the tape alphabet at the start of the construction and turn it into a constant of the semantics for this particular device A. An alternative is to work with the infinite alphabet Σ as introduced in the previous section. The fact that in the transition many tape symbols are allowed which won't be used by the actual machine has no influence on the eventual semantics according to our construction. Finally one can restrict oneself to Turing automata over a fixed alphabet; as is well known this can be done without loss of generality. Note that a similar problem with the state alphabet doesn't arise, due to the fact that we may restrict ourselves to those strings which contain exactly one symbol from Q.

The observation that compositionality for a well understood semantical interpretation can be obtained by modifying the syntax is not new. In the dissertation of Theo Janssen mentioned before [5] there are presented a number of examples in early extensions of Montague grammar where authors have deviated from the compositionality principle mainly on behalf of syntactic prejudice; the semantic phenomena they were after were easily brought within the scope of the compositionality principle by having another look at the syntax.

The reconstruction of a compositional semantics for Turing machines in this note illustrates that similar prejudices may lead to misconceptions in programming language semantics as well.

The positive result of our exercise does not mean that the compositionality achieved will be very helpful in understanding the behavior of Turing machines. The transitive closure operator easily will obliterate all structure of the union operator on the instructions. The complexity is caused by the fact that union and transitive closure are non-commuting operators, but that is a situation which is not unusual in mathematics.

Only for programs which are structured in some very restricted way this union may sort of commute with the transitive closure and reappear in the shape of a meaningful operation like functional composition, nondeterministic choice, iteration or one of the other composition operators which are well known in the theory of semantics of programming languages. See for example the description of the combination of Turing machines, called *machine schemata* in section 4.3 of [7]. What we have achieved therefore by our analysis is that we may start to understand why such restrictions on the program structure are required and what kind of restrictions we should aim for. The real work, however still has to be done. Clearly an interesting research topic for the next 25 years.

References

[1] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall Series in Automatic Computation 1976.

[2] Report on the Busy beaver competition 1982/83 distributed at the 6th conference on theoretical computer science, Jan 05-07, 1983, Dortmund. See also EATCS bulletin vol. 19, Feb 1983, pp. 77

[3] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley (1979)

[4] Janssen, T.M.V., *Foundations and applications of Montague grammar, Part 1: Philosophy, framework, computer science*, CWI Tract 19 (1986)

[5] Janssen, T.M.V., *Foundations and applications of Montague grammar, Part 2: Applications to natural language*, CWI Tract 28 (1986)

[6] Janssen, T.M.V. and van Emde Boas, P., *Some observations on compositional semantics*, in Kozen, D., ed., Logic of programs, Proceedings 1981, Springer Lecture notes in computer science 131 (1982) pp. 137-149

[7] Lewis, H.R. and Papadimitriou, C.H., *Elements of the Theory of Computation*, Prentice-Hall (1981)

[8] Rogers, H. Jr, *Theory of Recursive Functions and Effective Computability*, Mc Graw-Hill Series in Higher Mathematics (1967)

[9] Turing, A.M., *On computable numbers, with an application to the entscheidungsproblem*, Proc. London Math. Soc. ser. 2, 42 (1936) 230-265

[10] van Emde Boas, P. *Machine models and simulations*, to appear in J. van Leeuwen (ed.), Handbook of theoretical computer science, North Holland Publ. Comp. 1989. Preprint: rep. ITLI-CS-89-02.

COOPERATING-PROOFS FOR DISTRIBUTED PROGRAMS

WITH MULTI-PARTY INTERACTIONS

by

Nissim Francez

Computer Science Department

Technion - Israel Institute of Technology

Haifa 32000, Israel

Abstract:

The paper presents a proof-system for partial-correctness assertions for a language for distributed programs based on *multi-party interactions* as its interprocess communication and synchronization primitive. The system is a natural generalization of the *cooperating proofs* introduced for partial-correctness proofs of CSP programs.

1. INTRODUCTION

The proof-theoretical notion of *cooperating-proofs* was introduced in [AFR80] as a tool for the verification of CSP distributed programs [Ho78], based on synchronous *send-receive* (also known as *handshaking* and *rendezvous*) interprocess communication. It follows the structure of two-leveled proof systems as introduced in [OG76] in the context of shared-variables concurrency. A closely-related proof system is presented in [LG81].

Since its introduction, the notion of cooperating-proofs was adapted to several contexts, capturing interprocess communication constructs of increasing complexity and of varieties of structure. It was applied to remote procedure calls in [GRR82] for the DP language [Bh78], to ADA's *rendezvous* in [GR84] and [BM82], to *monitors* in [GR86], and to *scripts* (a communication abstraction mechanism) in [FHT86]. Very good surveys of cooperating-proofs are [dR85] and [HR86]. Recently, the proof method was extended to *dynamic process creation* in [B87], and to *dynamic creation and destruction* in [FF87].

In all the above extensions, interprocess communication can be classified as *point-to-point* communication, involving two processes: a *sender* and a *receiver*. Multiple engagement *is* possible in some variants of a rendezvous only due to *nesting*. In recent developments of languages for distributed computing a new family of communication structures has evolved, referred to generically as *multi-party interactions*, involving a simultaneous activity of an arbitrary collection of processes, usually fixed in advance (not varying during computation). Several such proposals are: *scripts* [FHT86], *joint-actions* [BKs83], *teams* and *interactions* in *Raddle* [Fo87], *shared-actions* [RM87] and *compacts* [Ch87]. These primitives enjoy a higher level of abstraction, hiding a lot of low-level details and encourage modular programming and design.

In [EFK88] these proposals are classified into *communication primitives*, and *communication abstractions*.

In this paper, we focus on the former and propose an extension of *cooperating-proofs* to the language IP ($Interacting\ \ Processes$) which uses the multi-party (synchronous) interaction as its sole interprocess communication and synchronization primitive. This language was introduced in [AF87], where an adequate notion of *fairness* for multi-party interactions is proposed.

While we are witnessing an increasing use of this family of constructs, very little has been done on extending the verification techniques to apply to these constructs. The main point of this paper is showing that cooperating-proofs can be very smoothly extended to the multi-party interactions, with some natural generalization of the concepts involved, but with no need for any essentially different proof-theoretic machinery. It is hoped that these generalizations, which enable formal proofs of (partial-) correctness of programs formulated by means of multi-party

interactions, will encourage an even more extensive usage of such primitives.

Cooperating-proofs belong to the two-leveled proof-systems, in which local proofs for the distinct processes are designed at the first stage, employing some assumptions about the environment's behavior; then, at the second stage, the local proofs are confronted for mutual consistency of all the assumptions made. As is known by now, such two-leveled proofs, which are semi-compositional, are the best possible without extending the correctness properties and the specification language beyond the usual partial-correctness assertions $\{p\}S\{q\}$, where p and q are state predicates.

In Section 2 the IP language and its formal operational semantics are introduced. In Section 3 local proofs are described, while the global *cooperation test* is presented in Section 4.

2. THE LANGUAGE IP (Interacting Processes)

In this section we present a simple mini-programming language, called *IP (Interacting Processes)*, first presented in [AF87]. The language is an abstraction and simplification of languages having multi-party interaction as their interprocess communication and synchronization primitive, and is suitable for focusing on cooperating proofs for partial correctness. In particular, multi-party interactions can be used as *guards*, thereby generalizing both Dijkstra's original guarded commands [Dij 76], which have only boolean guards, and CSP [Ho 78], using synchronous binary communication as guards. This kind of a language was already mentioned in [AFK 87], without details and formal semantics.

A program $P :: [P_1 \| \cdots \| P_n]$ consist of a *concurrent composition* of $n \geq 1$ (fixed n) *processes*, having *disjoint* local states (i.e., no shared variables). A *process* P_i, $1 \leq i \leq n$ consists of a statement S, where S may take one of the following forms:

skip: A statement with no effect on the state.

assignment $x:=e$: Here x is a variable local to P_i and e is an expression over P_i's local state. Assignments have their usual meaning of state transformation.

interaction $in\,[\bar{v}:=\bar{e}]$: Here in is the *interaction name* and $[\bar{v}:=\bar{e}]$ is an optional parallel assignment constituting a *local interaction-body* (where an empty body appears as $in\,[]$). All variables in $\bar{v}$ are local to P_i and different

from each other. The expressions $\bar{e}$ may involve variables *not local* to P_i (belonging to other parties of that interaction). The *participants* of an interaction in, denoted by PA_{in}, consists of all processes which syntactically refer to in in their program. When a process arrives (during execution) to a local interaction-box, it is said to *ready* the corresponding interaction. An interaction in is *enabled* only when *all* its participants have arrived to an interaction point involving in, at which point it can be executed. Its execution implies the execution of all the parallel assignments of all the local interaction-boxes of all participants. Every reference in the right hand side of an assignment in one local box to a variable belonging to another another participating process always means using the initial value, i.e. the one determined by the state at the time the interaction started. In [EFK88] some more complicated interaction-bodies are considered.

Thus, an interaction synchronizes all its participants, and all the bodies are executed in parallel. Upon termination of *all* bodies, each process resumes its local thread of control. Note that if the body $\bar{v}:=\bar{e}$ is empty, the effect is pure synchronization.

sequential composition $S_1; S_2$: First execute S_1; if and when it terminates, execute S_2. We freely use $S_1; \cdots ; S_k$ for any $k \geq 2$.

nondeterministic selection $[\underset{k=1,m}{[]} b_k; in_k[\bar{v}_k:=\bar{e}_k] \rightarrow S_k]$: Here $b_k; in_k[\bar{v}_k:=\bar{e}_k]$ is a *guard*, composed of two parts. The part b_k is a boolean expression over the local state of P_i. The part $in_k[\bar{v}_k:=\bar{e}_k]$ is an *interaction guard*. S_k is any statement. When such a statement is evaluated in some state, the k'th guard is *open* if b_k is true in that state and is readied at that stage. In general, several interactions may be readied by such a statement and are said to be in *conflict*. Executing this kind of statement involves the following steps: evaluate all boolean parts to determine the collection of open guards. If this collection is empty the statement *fails*. Otherwise a guard with an enabled interaction is passed (simultaneously with the execution of all the other bodies in the other parties) and S_k is executed.

nondeterministic iteration $*[\underset{k=1,m}{[]} b_k; in_k\ [\bar{v}_k:=\bar{e}_k] \rightarrow S_k]$: Similar to the choice, but execution terminates once no open guards exist, and the whole procedure is repeated after each execution of a guarded command.

In both the selection and iteration constructs, identically true guards may be omitted. Note that nested concurrency is excluded by this definition.

We now turn to formal definitions of the operational semantics, based on Plotkin's transition scheme [Pl 83]. In [AF87] two different semantics are defined: *serialized* and *overlapping* (compare with a similar distinction in [GFK 84] and [BKS 85]). The distinction between them is crucial to understanding the *hyperfairness* notion suggested in that paper. Since we are interested here only in partial correctness, only the serialized semantics is presented (they are equivalent in this respect, differing on liveness properties only). Throughout we assume some *interpretation* over which computations occur, and leave it implicit. The central characteristic of the semantics is that actions and interactions take place one at a time. A *configuration* $<[S_1 \| \cdots \| S_n], \sigma>$ consists of a concurrent program and a *global* state, assigning values to all variables. A configuration represents an intermediate stage in a computation where S_i is the rest of the program that process P_i has still to execute, while σ is the *current* state at that stage. We stipulate (for facilitating the definition) an *empty* program E (not in the IP language) satisfying the identities $S; E = E; S = S$ for every S. A configuration $<[E \| \cdots \| E], \sigma>$ is a *terminal* configuration. For a state σ, we use the usual notions of a *variant* $\sigma[a/x]$ and $\sigma[\bar{a}/\bar{x}]$.

We now define the *(serialized) transition* relation $\rightarrow$ among configurations.

$$<[S_1 \| \cdots \| S_i \| \cdots \| S_n], \sigma> \rightarrow <[S_1 \| \cdots \| E \| \cdots \| S_n], \sigma> \tag{1}$$

for any $1 \le i \le n$ iff $S_i = skip$, or $S_i = *[\underset{j=1,n_i}{\square} b_j; in_j[\bar{v}_j := \bar{e}_j] \rightarrow T_j]$ and $\neg \underset{j=1,n_i}{\vee} b_j$ holds in σ.

$$<[S_1 \| \cdots \| S_i \| \cdots \| S_n], \sigma> \rightarrow <[S_1 \| \cdots \| E \| \cdots \| S_n], \sigma[\sigma[e]/x]> \tag{2}$$

for any $1 \le i \le n$ iff $S_i = (x := e)$.

$$\begin{aligned} &<[S_1 \| \cdots \| S_{i_1-1} \| S_{i_1} \| S_{i_1+1} \| \cdots \| S_{i_k-1} \| S_{i_k} \| S_{i_k+1} \| \cdots \| S_n], \sigma> \rightarrow \\ &<[S_1 \| \cdots \| S_{i_1-1} \| S'_{i_1} \| S_{i_1+1} \| \cdots \| S_{i_k-1} \| S'_{i_k} \| S_{i_k+1} \| \cdots \| S_n], \sigma'> \end{aligned} \tag{3}$$

iff the following holds: There is an interaction in with a set of participants $PA_{in} = \{P_{i_1}, \cdots, P_{i_k}\}$ (for some $1 \le k \le n$), and for every i s.t. $P_i \in PA_{in}$ one of the following conditions holds:

(a) $S_i = in[\bar{v}_i := \bar{e}_i]$ and $S'_i = E$

(b) $S_i = [\underset{j=1,n_i}{\square} b_j; in_j[\bar{v}_j := \bar{e}_j] \rightarrow T_j]$ and there exists some j, $1 \le j \le n_i$ s.t. b_j holds in σ, $in_j = in$ and $S'_i = T_j$

(c) $S_i = *[\underset{j=1,n_i}{\square} b_j; in_j[\bar{v}_j := \bar{e}_j] \rightarrow T_j]$ and there exists some j, $1 \le j \le n_i$ s.t. b_j holds in σ, $in_j = in$, $S'_i = T_j; S_i$.

Finally, for all these cases, $\sigma'=\sigma[\sigma[\bar{e}]/\bar{v}]$, with $\bar{v}=\bigcup_{P_i\in PA_{in}} \bar{v}_i$ and $\bar{e}=\bigcup_{P_i\in PA_{in}} \bar{e}_i$.

For any $1\le i\le n$, if

$$< [S_1 \| \cdots \| S_i \| \cdots \| S_n], \sigma > \to < [S_1 \| \cdots \| S'_i \| \cdots \| S_n], \sigma' > \quad (4)$$

then

$$< [S_1 \| \cdots \| S_i ; T_i \| \cdots \| S_n], \sigma > \to < [S_1 \| \cdots \| S'_i ; T_i \| \cdots \| S_n], \sigma' >$$

For this semantics, we define the following notions.

Definition:

(1) A *(serialized) computation* π of P on an initial state σ is a maximal (finite or infinite) sequence of configurations C_i, $i\ge 0$, such that:

(a) $C_0=< P, \sigma >$.

(b) For all $i\ge 0$, $C_i \to C_{i+1}$.

(2) The computation π *terminates* iff it is finite and its last configuration is terminal, and π *deadlocks* iff it is finite and its last configuration is *not* terminal.

(3) An interaction in is *enabled* in a configuration C iff C has one of the forms in clause (3) of the definition of '$\to$' and all the conditions are satisfied for in.

(4) Two interactions in_1 and in_2 are in *conflict* in a configuration C iff both interactions are enabled in C and they have non-disjoint set of participants, i.e. $PA_{in_1} \cap PAin_2 \ne \emptyset$.

Based on this definitions it is now possible to interpret partial-correctness assertions in the usual way. In the next two sections we present the extension of cooperating proofs to the language IP.

3. LOCAL PROOF-OUTLINES

The basic components of a proof are represented as *proof-outlines*, which were introduced in [OG76]. Our use of them is similar to that in [AFR80]. proof-outlines constitute the local part of a proof, in which separate reasoning about each process is carried out. proof-outlines are to be contrasted with each other at the second stage of a

correctness proof. (A formulation in terms of *proofs from assumptions* appears in [Ap84].) The proof-outlines for IP satisfy the usual partial-correctness axioms and rules, and in addition the following *interaction-box axiom (iba).*

(iba) $\{p\}$ in $[\overline{v}:=\overline{e}]\{q\}$, for *arbitrary* local assertions p and q.

This axiom is the natural generalization of the i/o axioms in [AFR80, LG81] and their counter-parts in the more complicated applications of cooperative proofs mentioned in the introduction.

In so much as the i/o axioms constitute an *assumption* about the state of the communication partner (in some matching communication), the interaction-box axiom may be considered to constitute a *joint-assumption* about the preinteraction states of *all* the participants of some collection of matching local interaction-boxes. The collection of all such joint-assumptions made by all members of PA_{in} are confronted simultaneously in the *cooperation-test* described in the next section. This is the proof-theoretical counterpart of the synchronized nature of a multi-party interaction.

Example: As a simple example, we consider the following applications of the iba axiom.

$$P :: [P_1 :: in\ [x:=x+y+z] \parallel P_2 :: in\ [y:=x+y+z] \parallel P_3 :: in\ [z:=x+y+z]].$$

Each of the following is a valid proof-outline:

$$\{x \geq 0\}\ in\ [x:=x+y+z]\{x \geq 0\} \text{ for } P_1,$$
$$\{y \geq 0\}\ in\ [y:=x+y+z]\{y \geq 0\} \text{ for } P_2,$$
$$\{z \geq 0\}\ in\ [z:=x+y+z]\{z \geq 0\} \text{ for } P_3.$$

Thus, in the first one, we conclude locally for P_1 a postcondition $x \geq 0$ based on its own precondition and an assumption about the preconditions of P_2 and P_3, about the values of y and z. As it happens, the conjunction of the respective preconditions $y \geq 0$ and $z \geq 0$ indeed satisfies the assumption, as would be revealed during the cooperation test.

As is usual for partial correctness proofs in distributed programs, another kind of an assumption made by a process is that a certain interaction will *never occur* (compare with [AFR80] and [Ap84] for the same phenomenon in *send-receive* communication). This can be seen in the following simple example.

Example: Consider the program Q in Figure 1. Note that $PA_{in_2} = \{P_1, P_2, P_3\}$, *in spite* of the identically $false$ guards (introduced for simplicity, to avoid local computation) of in_2 in P_2 and P_3.

As a part of a proof of $\{true\}Q\{x=y=0\}$, we might have a proof-outline for P_1 with a postcondition $\{x=0\}$. However, since x is modified both in the body of in_1 and in the body of in_2, to preserve the sequential

Q :: [P_1:: [$in_1[x:=y]$ → *skip* □ $in_2[x:=z]$ → *skip*]
    ||
    P_2:: $y:=0$;[*true* → $in_1[]$ □ *false* → $in_2[]$]
    ||
    P_3:: $z:=1$; [*true* → $in_1[]$ □ *false* → $in_2[]$]
    ].

Figure 1: Example program Q

nondeterministic branching rule an assumption is needed also about the local interaction-box $in_2[x:=z]$, even though this interaction will never occur. We obtain the following (Figure 2) proof-outline for P_1. The interaction in_2 will pass the cooperation-test *vacuously*, by having a *false* precondition in any matching collection of local interaction-boxes.

4. GLOBAL INVARIANTS AND THE COOPERATION TEST

Let P be an IP program and in an interaction in P. Our most basic aim, after having separate proof-outlines for each P_i in P, is to establish

$$(*)\ \{\bigwedge_{P_j \in PA_{in}} pre_j\} \left[in[\bar{v}_{i_1}:=\bar{e}_{i_1}] \,||\, \cdots \,||\, in[\bar{v}_{i_k}:=\bar{e}_{i_k}] \right] \{\bigwedge_{P_j \in PA_{in}} post_j\}.$$

Here $\{pre_j\}\ in[\bar{v}_j:=\bar{e}_j]\{post_j\}$ is taken from the local proof-outline of P_j, $P_j \in PA_{in}$, for a syntactically-matching collection of local interaction-boxes. The correctness assertion (*) captures the operational synchronous nature of an interaction.

P_1:: {*true* }
    [$in_1[x:=y]\{x=0\}$ → *skip* $\{x=0\}$
    □
    $in_2[x:=z]\{x=0\}$ → *skip* $\{x=0\}$]
    ] $\{x=0\}$

Figure 2: A proof-outline for P_1

The first step is to introduce the *(global) interaction axiom (gia)*, the natural generalization of the *communication axiom* of [AFR80] and [LG81].

$$(gia)\ \{p_{\bar{e}}^{\bar{v}}\}[\ \underset{P_j \in PA_{in}}{\|}\ in[\bar{v}_j := \bar{e}_j]]\{p\},\ \text{where } \bar{v} = \underset{P_j \in PA_{in}}{\cup} \bar{v}_j \text{ and } \bar{e} = \underset{P_j \in PA_{in}}{\cup} \bar{e}_j\ .$$

This axiom captures the operational semantics of an interaction as a parallel (atomic) execution of all the parallel assignments is the local interaction-boxes. This is the usual axiom for multiple assignments [LG81], with two levels of multitude being collapsed: One within a local interaction-box and another among concurrent local interaction-boxes. The syntactic constraint of all variables in $\bar{v}$ being different follows from the formation rules of local interaction-boxes in IP and from process state disjointness.

Example: Returning to the program P in a previous example, an application of the interaction-axiom and the rule of consequence immediately yield

$$(**)\ \{x \geq 0 \wedge y \geq 0 \wedge z \geq 0\}\left[\ in[x := x+y+z]\ \|\ in[y := x+y+z]\ \|\ in[z := x+y+z]\ \right]\{x \geq 0 \wedge y \geq 0 \wedge z \geq 0\}.$$

We might also prove a stronger partial-correctness assertion, about P, with the same precondition and the postcondition $x=y=z \geq 0$. To obtain this proof, we have to appeal also to the usual *substitution rule* of [Go75] (see also [Ap84] for its use for cooperation proofs). The problem is that we need to "freeze" initial values of variables, to which local interaction-boxes have access, but local assertions do not. We modify the proof outlines for that example.

The stronger proof-outlines for the program P above are:

$$\{x = a \geq 0\}\ in[x := x+y+z]\{x = a+b+c \geq 0\}\ \ for\ P_1,$$
$$\{y = b \geq 0\}\ in[y := x+y+z]\{y = a+b+c \geq 0\}\ \ for\ P_2,$$
$$\{z = c \geq 0\}\ in[z := x+y+z]\{z = a+b+c \geq 0\}\ \ for\ P_3.$$

Since

$$x = a \geq 0 \wedge y = b \geq 0 \wedge z = c \geq 0\ \Rightarrow\ x+y+z=a+b+c \geq 0$$

An application of the global interaction-axiom and the consequence rule yields

$$\{x = a \geq 0 \wedge y = b \geq 0 \wedge z = c \geq 0\}\ P\ \{x=a+b+c=y=z \geq 0\}.$$

By weakening the postcondition, we finally get

$$\{x = a \geq 0 \wedge y = b \geq 0 \wedge z = c \geq 0\}\ P\ \{x=y=z \geq 0\}.$$

Since none of a, b and c is free either in P or in the postcondition, we can substitute $a \mid x$, $b \mid y$, $c \mid z$, to obtain

$$\{x \geq 0 \wedge y \geq 0 \wedge z \geq 0\}\ P\ \{x=y=z \geq 0\}.$$

Without substitution, we could obtain an even stronger postcondition, "remembering" the summation effect of the program.

□

However, as in the case of communicating processes [AFR80], things are a little more complicated. The main problem is the identification of semantically-matching interactions. So, similarly to the communicating processes case, we introduce *auxiliary variables* [OG76, AFR80], that carry the additional information needed to express the conditions for such a semantic matching, which happen not to be present in the state. In addition, we introduce a *global invariant* and *bracketed sections*, where the invariant has to hold before and after every execution of a semantically-matching tuple of bracketed sections. The global invariant may refer to the variable in *all* processes. While its main role is the one stated above, in actual proofs it can be used to propagate *any* global information.

We now slightly extend the bracketing notion of communicating processes.

Definition:

(1) A process P_i is *bracketed* iff the brackets ' $<$ ' and ' $>$ ' is interspersed in its body in such a way that each *bracketed section* $< S >$ is of the form S_1; $in[\overline{v}:=\overline{e}]$; S_2, for some local S_1 and S_2 (i.e., without interaction boxes, possibly empty).

(2) A collection of local interaction-boxes $\{in_i[\overline{v}_i:=\overline{e}_i] \mid i \in A\}$ *syntactically-match* iff for some interaction in, $\bigwedge_{i \in A} in_i=in$ holds, where $in_i[\overline{v}_i:=\overline{e}_i]$ occurs in process P_i. Furthermore, $A=PA_{in}$.

(3) A collection $<S_i>$, $i \in A$, of bracketed sections *syntactically match* iff the collection of respective local interaction -boxes occurring within them syntactically match.

(4) An *outline section* $\{p\}<S>\{q\}$ is a bracketed section with its precondition and postcondition taken from a local proof-outline of some process. Outline sections syntactically-match when their respective bracketed sections match.

□

As for communicating processes, the bracketing suggest a slightly different operational semantics, where the grain of atomicity is a semantically matching pair of bracketed sections. As far as partial correctness is concerned, this is equivalent to the original semantics, and enables updating auxiliary variables together with visible effects in an interaction. For an explanation of this equivalence in term of temporal logic see [KP87]. This approach extends to IP as well, taking as atomic steps semantically matching collections of local interaction-boxes.

We now introduce a global invariant I, to be preserved by a program P when executed with this coarser grain of interleaving, induced by the bracketed sections. The invariant has free variables whose values are modifiable only *within* bracketed sections.

Definition:

(1) A syntactically matching collection of outline sections $\{pre_i\} <S_i> \{post_i\}$, $1 \le i \le n$ *cooperates* w.r.t. the invariant I iff $\{ \bigwedge_{i=1,m} pre_i \wedge I\} <S_1 \| \cdots \| S_n> \{ \bigwedge_{i=1,m} post_i \wedge I\}$ can be proved.

(2) For an IP program $P :: [\|_{i=1,n} P_i]$, proof-outlines $\{p_i\} P_i \{q_i\}$, $1 \le i \le n$ *cooperate* w.r.t. I iff every syntactically matching collection of outline sections cooperate w.r.t. I.

□

To establish the condition in (1), one can use the global interaction-axiom and *formation rules* similar to the ones in [AFR80] or [Ap84].

We now can introduce the usual *concurrent composition (cc)* meta-rule, which is the *same* as for communicating processes., but with the extended definition of cooperation among proofs.

Let $P :: [P_1 \| \cdots \| P_n]$ be an IP program and let $\{p_i\}P_i\{q_i\}$, $1 \le i \le n$, be valid local proof-outlines.

$$(cc) \quad \frac{\{p_i\}P_i\{q_i\},\ 1 \le i \le n,\ cooperate\ w.r.t.\ I}{\{ \bigwedge_{i=1,n} p_i \wedge I\}\ P\ \{ \bigwedge_{i=1,n} q_i \wedge I\}}.$$

It is interesting to note that the concurrent composition rule remains intact, and is valid for many situations of cooperation of local outline. It might be interesting to find an abstract characterization of cooperation that will have (cc) as a valid rule. No such characterization exists to date.

Finally, one uses the usual *auxiliary variables* rule ([OG76], [AFR80]) to get rid of the auxiliary variables and assingments introduced for the sake of the proof only.

Acknowledgements

I wish to thank Shmuel Katz for useful discussions. This work was initiated during a summer visit to MCC, July-August 1987 and then continued during consulting with MCC. The continuation at the technion was also partially supported by the Foundation for Research in Electronics, Computers and Communications administered by the Israeli Academy of Sciences and Humanities and by the Fund for the Promotion of Research in the Technion.

References:

[AF 87] P.C. Attie, N. Francez: "Fairness and hyperfairness in multiparty interactions", MCC-STP TR 356-87, August 1987. A revised and corrected version, with Orna Grumberg as an additional coauthor, was submitted for publication.

[AFK 87] K.R. Apt, N. Francez, S. Katz: "Appraising Fairness in Distributed Languages", proc. 14th ACM-POPL symposium, Munich, Germany, Jan. 1987.

[AFR 80] K.R. Apt, N. Francez, W.P. de Roever: "A proof system for communicating sequential processes", ACM-TOPLAS 2, 3: 359-380, July 1980.

[Ap 84] K.R. Apt: "Proving correctness of CSP programs - a tutorial", Proc. int. summer school "Control flow and data flow: concepts in distributed programming", Springer-Verlag (M. Broy - ed.), NATO ASI series F, 1984.

[dB 86] F.S. deBoer: "A proof rule for process-creation", proc. 3rd working conf. on Formal description of programming concepts, Ebberup, Denmark, August 1986.

[Bh 78] P. Brinch-Hansen: "Distributed Processes - a concurrent programming concept", CACM 21, 1978, pp. 934-941.

[BKS 83] R.J. Back, R. Kurki-Suonio: "Decentralization of Process Nets With Centralized Control", proc. 2nd ACM-PODC, Montreal, Canada, August 1983.

[BM 82] H. Barringer, I. Mearns: "Axioms and proof rules for Ada tasks", IEE PROC.m vol. 129, Pt. E, no. 2, March 1982.

[BKs 85] R.J. Back, R. Kurki-Suonio: "Serializability in Distributed Systems With Handshaking", TR 85-109, CMU, 1985.

[Ch 87] A. Charlesworth: "The Multiway Rendezvous", ACM-TOPLAS 9,2:350-366, July 1987.

[Dij 76] E.W. Dijkstra: **A Discipline of Programming,** Prentice-Hall, 1976.

[EFK 88] M. Evangelist, N. Francez, S. Katz: "Multi-party interactions for interprocess communication and synchronization", MCC/STP TR, 1988. To appear in IEEE-TES, 1989.

[FHT 86] N. Francez, B.T. Hailpern, G. Taubenfeld: "SCRIPT - A Communication Abstraction Mechanism and Its Verification", Science of Computer Programming 6,1, pp. 35-88, Jan. 1986.

[Fo 87] I.R. Forman: "On the Design of Large Distributed Systems", TR STP-098-86 (Rev. 1.0), MCC, Austin, TX, Jan. 1987. A preliminary version presented at the First International Conf. on Computer Languages, Miami, FL, October 1986.

[FF 88] L. Fix, N. Francez: "Proof rules for dynamic process creation and destruction", Technion, April 1988, submitted for publication.

[GFK 84] O. Grumberg, N. Francez, S. Katz: "Fair Termination of Communicating Processes", 3rd ACM-PODC Conference, Vancouver, BC, Canada, August 1984.

[Go 75] G.A. Gorelick: "A complete axiomatization system for proving assertions about recursive and nonrecursive programs", TR 76, Dept. of Computer Science, University of Toronto, 1975.

[GR 84] R. Gerth, W.P. deRoever: "A proof system for concurrent Ada programs", SCP 4, 1984, 159-204.

[GR 86] R. Gerth, W.P. deRoever: "Proving monitors revisited - a first step towards verifying object oriented systems", Fundamenta Informatica IX, 1986.

[GRR 82] R. Gerth, W.P. deRoever, M. Roncken: "Procedures and concurrency- a study in proof", proc. 5th ISOP, 1982. LNCS 137, Springer-Verlag, 1982.

[Ho 78] C.A.R. Hoare: "Communicating Sequential Processes", CACM 21,8, pp. 666-678, August 1978.

[HR 86] J. Hooman, W. P. deRoever: "The quest goes on - a survey of proofsystems for partial correctness for CSP", EUT TR 86-WSK-01, Eindhoven U., January 1986.

[KP88] S. Katz, D. Peled: "Interleaving sets temporal logic", proc. 6th ACM-PODC, Vancouver, BC, August 1987. To appear in TCS.

[LG 81] G. Levin, D. Gries: "Proof techniques for communicating sequential processes", ACTA INFORMATICA 15: 281-302, 1981.

[OG 76] S. Owicki, D. Gries: "Verifying properties of parallel programs: An axiomatic approach", CACM 19, 5: 279-286, August 1976.

[Pl 83] G.D. Plotkin: "An Operational Semantics for CSP", TC.2 working group conference on the formal description of programming concepts, Garmisch Partenkierchen, (D. Biorner, ed.), North Holland, 1983.

[dR85] W.P. deRoever: "The cooperation test: a syntax-directed verification method", in: **Logics and models of concurrent systems,** (K.R. Apt - ed.), Nato ASI series F vol. 13, Springer Verlag, 1985.

[RM 87] S. Ramesh, S.L. Mehndiratta: "A methodology for developing distributed programs", IEEE-TSE vol. SE-13,8: 967-976, Aug. 1987.

The processes of De Bakker and Zucker represent bisimulation equivalence classes

Rob van Glabbeek & Jan Rutten
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The basic fact expressed by the title is not difficult to prove, in a sense is well known, and has yet never been proved in public. Voilà three reasons for this contribution.

DEFINITION 1 (*LTS*): A *labelled transition system* is a triple $\mathcal{Q}=(S,A,\rightarrow)$ consisting of a set of *states* S, a set of *labels* A, and a *transition relation* $\rightarrow\subseteq S\times A\times S$. We shall write $s\xrightarrow{a}s'$ for $(s,a,s')\in\rightarrow$. A LTS is called *image finite* if for all $s\in S$ and $a\in A$ the set $\{s'\colon s\xrightarrow{a}s'\}$ is finite.

DEFINITION 2: Let $\mathcal{Q}=(S,A,\rightarrow)$ be a LTS. A relation $R\subseteq S\times S$ is called *a (strong) bisimulation* if it satisfies for all $s,t\in S$ and $a\in A$:

$$(sRt \wedge s\xrightarrow{a}s') \Rightarrow \exists t'\in S\ [t\xrightarrow{a}t' \wedge s'Rt'] \text{ and}$$

$$(sRt \wedge t\xrightarrow{a}t') \Rightarrow \exists s'\in S\ [s\xrightarrow{a}s' \wedge s'Rt'].$$

Two states are *bisimilar,* notation $s\leftrightarrow t$, if there exists a bisimulation relation R with sRt. The relation $\leftrightarrow$ is again a bisimulation. Note that bisimilarity is an equivalence relation on states.

DEFINITION 3 (Processes): Let the set of processes P be the unique complete metric space that satisfies the following reflexive equation:

$$P \cong \mathcal{P}_{closed}(A\times P).$$

Let d be the metric on P. The metric on $\mathcal{P}_{closed}(A\times P)$ is the Hausdorff metric d_H induced by the following metric on $A\times P$:

$$\bar{d}(<a_1,p_1>,<a_2,p_2>) = \begin{cases} 1 & \text{if } a_1\neq a_2 \\ \tfrac{1}{2}\cdot d(p_1,p_2) & \text{if } a_1=a_2. \end{cases}$$

The Hausdorff metric d_H is given, for every $X, Y \in \mathcal{P}_{closed}(A \times P)$, by

$$d_H(X,Y) = \max\{\sup_{x \in X}\{d(x,Y)\}, \sup_{y \in Y}\{d(y,X)\}\},$$

where $d(x,Z) = \inf_{z \in Z}\{\bar{d}(x,z)\}$ for every $Z \in \mathcal{P}_{closed}(A \times P)$ and $x \in A \times P$. (By convention $\sup \varnothing = 0$ and $\inf \varnothing = 1$.)

DEFINITION 4: Let $\mathcal{Q} = (S, A, \rightarrow)$ be an image finite LTS. We define a mapping $\mathfrak{M}: S \rightarrow P$ by

$$\mathfrak{M}[\![s]\!] = \{<a, \mathfrak{M}[\![s']\!]>: s \xrightarrow{a} s'\}.$$

Actually, the precise definition of $\mathfrak{M}$ is $\mathfrak{M}[\![s]\!] = i(\{<a, \mathfrak{M}[\![s']\!]>: s \xrightarrow{a} s'\})$, with $i: \mathcal{P}_{closed}(A \times P) \rightarrow P$ an isometry between $\mathcal{P}_{closed}(A \times P)$ and P, but for convenience we usually leave out isometry symbols. Remark that the isometry i is necessary to stay within well-founded set theory.

We can justify this recursive definition by taking $\mathfrak{M}$ as the unique fixed point (Banach's theorem) of a contraction $\Phi: (S \rightarrow P) \rightarrow (S \rightarrow P)$, defined by

$$\Phi(F)(s) = \{<a, F(s')>: s \xrightarrow{a} s'\}.$$

The fact that Φ is a contraction can be easily proved. The closedness of the set $\Phi(F)(s)$ is an immediate consequence of the image finiteness of $\mathcal{Q}$: Consider a Cauchy sequence $(<a_i, F(s_i)>)_i$ in $\Phi(F)(s)$. From the definition of the metric on $A \times P$ it follows that there exist $\bar{a} \in A$ and $I \in \mathbb{N}$ such that $a_i = \bar{a}$ for all $i > I$. Because $\mathcal{Q}$ is image finite there exists $\bar{s}$ with $s_i = \bar{s}$ for infinitely many i's. Thus the entire sequence $(<a_i, F(s_i)>)_i$ has $<\bar{a}, F(\bar{s})> \in \Phi(F)(s)$ as its limit.

THEOREM 1: *Let $\mathcal{Q} = (S, A, \rightarrow)$ be an image finite LTS. Then:*

$$\forall s, t \in S\ [s \leftrightarrow t \Leftrightarrow \mathfrak{M}[\![s]\!] = \mathfrak{M}[\![t]\!]].$$

PROOF: Let $s, t \in S$.

$\Leftarrow$:

Suppose $\mathfrak{M}[\![s]\!] = \mathfrak{M}[\![t]\!]$. We define a relation $\equiv \subseteq S \times S$ by

$$s' \equiv t' \Leftrightarrow \mathfrak{M}[\![s']\!] = \mathfrak{M}[\![t']\!].$$

From the definition of $\mathfrak{M}$ it is straightforward that $\equiv$ is a bisimulation relation on S: Suppose $s' \equiv t'$ and $s' \xrightarrow{a} s''$; then $<a, \mathfrak{M}[\![s'']\!]> \in \mathfrak{M}[\![s']\!] = \mathfrak{M}[\![t']\!]$; thus there exists $t'' \in S$ with $t' \xrightarrow{a} t''$ and $\mathfrak{M}[\![s'']\!] = \mathfrak{M}[\![t'']\!]$, that is, $s'' \equiv t''$. Symmetrically, the second property of a bisimilation relation holds. From the hypothesis we have $s \equiv t$. Thus we have $s \leftrightarrow t$.

$\Rightarrow$:

Let $R \subseteq S \times S$ be a bisimulation relation with sRt. We define

$$\epsilon = \sup_{s', t' \in S}\{d(\mathfrak{M}[\![s']\!], \mathfrak{M}[\![t']\!]): s'Rt'\}.$$

We prove that $\epsilon = 0$, from which $\mathfrak{M}[\![s]\!] = \mathfrak{M}[\![t]\!]$ follows, by showing that $\epsilon \leq \frac{1}{2} \cdot \epsilon$. We prove for all s', t' with $s'Rt'$ that $d(\mathfrak{M}[\![s']\!], \mathfrak{M}[\![t']\!]) \leq \frac{1}{2} \cdot \epsilon$. Consider $s', t' \in S$

with $s'Rt'$. From the definition of the Hausdorff metric on P it follows that it suffices to show

$$d(x,\mathfrak{M}[\![t']\!]) \leqslant \tfrac{1}{2}\cdot\epsilon \quad\text{and}\quad d(y,\mathfrak{M}[\![s']\!]) \leqslant \tfrac{1}{2}\cdot\epsilon$$

for all $x \in \mathfrak{M}[\![s']\!]$ and $y \in \mathfrak{M}[\![t']\!]$. We shall only show the first inequality, the second being similar. Consider $<a,\mathfrak{M}[\![s'']\!]>$ in $\mathfrak{M}[\![s']\!]$ with $s' \xrightarrow{a} s''$. Because $s'Rt'$ and $s' \xrightarrow{a} s''$ there exists $t'' \in S$ with $t' \xrightarrow{a} t''$ and $s''Rt''$. Therefore

$$\begin{aligned} d(<a,\mathfrak{M}[\![s'']\!]>,\ \mathfrak{M}[\![t']\!]) &= d(<a,\mathfrak{M}[\![s'']\!]>,\ \{<\bar{a},\mathfrak{M}[\![\bar{t}]\!]>:\ t' \xrightarrow{\bar{a}} \bar{t}\}) \\ &\leqslant [\text{ we have: } d(x,Y) = \inf\{d(x,y): y \in Y\}\ ] \\ &\quad\ d(<a,\mathfrak{M}[\![s'']\!]>,\ <a,\mathfrak{M}[\![t'']\!]>) \\ &= \tfrac{1}{2}\cdot d(\mathfrak{M}[\![s'']\!],\ \mathfrak{M}[\![t'']\!]) \\ &\leqslant [\text{ because } s''Rt''\ ]\ \tfrac{1}{2}\cdot\epsilon. \quad \square \end{aligned}$$

Next we will generalise theorem 1 to the case that $\mathcal{Q}$ is not required to be image finite. For this purpose we will work in Aczels universe of non-well-founded sets. This universe is an extension of the Von Neuman universe of well-founded sets, where the axiom of foundation (every chain $x_0 \ni x_1 \ni \cdots$ terminates) is dropped. Instead an anti-foundation axiom (AFA) is adopted, saying that systems of equations like the one in definition 4 have unique solutions. Let $\mathcal{V}$ be this universe. In $\mathcal{V}$ there exists a unique complete metric space P satisfying

$$P = \mathcal{P}_{closed}(A \times P).$$

This space can be regarded as a canonical representative of the space from definition 3 in the universe of non-well-founded sets. It can be obtained from any constructed solution of the domain equation in definition 3 by means of projection. Since this canonical representative contains non-well-founded sets indeed, it can not be found in the Von Neuman universe.

We can now extend definition 4 with image infinite LTSs.

DEFINITION 5: Let $\mathcal{Q}=(S,A,\rightarrow)$ be a LTS. We define a mapping $\mathfrak{M}: S \rightarrow \mathcal{V}$ by

$$\mathfrak{M}[\![s]\!] = \{<a,\mathfrak{M}[\![s']\!]>:\ s \xrightarrow{a} s'\}.$$

If $\mathcal{Q}$ is not image finite, $\mathfrak{M}[\![s]\!]$ for $s \in S$ may be outside P.

THEOREM 2: *Let $FSA = (S,A,\rightarrow)$ be a LTS. Then:*

$$\forall s,t \in S\ [s \leftrightarrow t \Leftrightarrow \mathfrak{M}[\![s]\!] = \mathfrak{M}[\![t]\!]].$$

PROOF: This theorem follows immediately from the categorical considerations in Aczels lecture notes on non-well-founded sets. Below we provide a direct non-categorical proof.

$\Leftarrow$:

Exactly as before.

$\Rightarrow$:

Let $\mathfrak{M}^*: S \rightarrow \mathcal{V}$ denote the unique solution of

$$\mathfrak{M}^*[\![s]\!] = \{<a, \mathfrak{M}^*[\![r']\!]>: \exists r \in S\ [r \leftrightarrow s \wedge r \xrightarrow{a} r']\}.$$

As for $\mathfrak{M}$ it follows from AFA that such a unique solution exists. Since $\leftrightarrow$ is an equivalence relation it follows that

$$s \leftrightarrow t \ \Rightarrow \ \mathfrak{M}^*[\![s]\!] = \mathfrak{M}^*[\![t]\!]. \qquad (*)$$

Hence it remains to be proven that $\mathfrak{M}^* = \mathfrak{M}$. This can be done by showing that $\mathfrak{M}^*$ satisfies the equations $\mathfrak{M}[\![s]\!] = \{<a, \mathfrak{M}[\![s']\!]>: s \xrightarrow{a} s'\}$, which have $\mathfrak{M}$ as unique solution. So it has to be established that

$$\mathfrak{M}^*[\![s]\!] = \{<a, \mathfrak{M}^*[\![s']\!]>: s \xrightarrow{a} s'\}.$$

The direction "$\supseteq$" follows directly from the reflexivity of $\leftrightarrow$. For "$\subseteq$", suppose $<a, X> \in \mathfrak{M}^*[\![s]\!]$. Then $\exists r, r'$: $r \leftrightarrow s$, $r \xrightarrow{a} r'$ and $X = \mathfrak{M}^*[\![r']\!]$. Since $\leftrightarrow$ is a bisimulation, $\exists s'$: $s \xrightarrow{a} s'$ and $r' \leftrightarrow s'$. Now from (*) it follows that $X = \mathfrak{M}^*[\![r']\!] = \mathfrak{M}^*[\![s']\!]$. Therefore $<a, X> \in \{<a, \mathfrak{M}^*[\![s']\!]>: s \xrightarrow{a} s'\}$, which had to be established.

Refinement in branching time semantics

R.J. van Glabbeek
W.P. Weijland
Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract: In this paper we consider branching time semantics for finite sequential processes with silent moves. We show that MILNER's notion of *observation equivalence* is not preserved under refinement of actions, even when no interleaving operators are considered; however, the authors' notion of *branching bisimulation* is.

INTRODUCTION

Virtually all semantic equivalences employed in theories of concurrency are defined in terms of *actions* that concurrent systems may perform (cf. [1-7]). Mostly, these actions are taken to be *atomic*, meaning that they are considered not to be divisible into smaller parts. In this case, the defined equivalences are said to be based on *action atomicity.*

However, in the top-down design of distributed systems it might be fruitful to model processes at different levels of abstraction. The actions on an abstract level then turn out to represent complex processes on a more concrete level. This methodology does not seem compatible with non-divisibility of actions and for this reason, PRATT [7], LAMPORT [4] and others plead for the use of semantic equivalences that are not based on action atomicity.

As indicated in CASTELLANO, DE MICHELIS & POMELLO [2], the concept of action atomicity can be formalised by means of the notion of *refinement of actions*. A semantic equivalence is *preserved under action refinement* if two equivalent processes remain equivalent after replacing all occurrences of an atomic action a by a more complicated process r(a). In particular, r(a) may be a sequence of two actions a_1 and a_2. An equivalence is strictly based on action atomicity if it is not preserved under action refinement.

In a previous paper [3] the authors argued that MILNER's notion of observation equivalence [5] does not respect the branching structure of processes, and proposed the finer notion of *branching bisimulation equivalence* which does. In this paper we moreover find, that observation equivalence is not preserved under action refinement, whereas branching bisimulation equivalence is.

1. PROCESS GRAPHS

As a simple model, let us represent a process by a *state transition diagram* or *process graph*. Such a graph has a node for every one of the possible states of the process, and has arrows between nodes to indicate whether or not a state is accessible from another. Furthermore, these arrows (directed edges) are labelled, with labels from $A \cup \{\tau\}$, where $A = \{a,b,c,...\}$ is some set of *observable signals*, and τ stands for a *silent step* (cf. [5]).

DEFINITION 1.1 A *process graph* is a connected, rooted, edge-labelled and directed graph.

In an edge-labelled graph, one can have more than one edge between two nodes as long as they carry different labels. A rooted graph has one special node which is indicated as the root node. Graphs need not be finite, but in a connected graph one must be able to reach every node from the root node by following a finite path. If r and s are nodes in a graph, then $r \to^a s$ denotes an edge from r to s with label a (it is also used as a proposition stating that such an edge exists). In this paper we limit ourselves to processes represented by finite, non-trivial process graphs. A graph is *finite* if it is acyclic and contains only finitely many nodes and edges; it is *trivial* if it contains no edges at all. The set of non-trivial, finite process graphs will be denoted by G.
In order to turn G into an algebraic structure, it is possible to define binary operators '+' and '$\cdot$' for alternative and sequential composition. For any two graphs g and h the process graph $(g + h)$ is obtained by simply identifying their root nodes, whereas $(g \cdot h)$ - often written as just (gh) - can be found by identifying the root node of h with all endnodes of g. Furthermore, constants from $A \cup \{\tau\}$ are interpreted as one-edge graphs, carrying the constant as their edge-label. The algebraic structure allows us to study equational theories that emerge from any defined equivalence on G. For instance, in branching time semantics, one often considers *observation congruence* (cf. MILNER [5]) - written as $\approx^c$ - as a deciding criterion for equality in observable behaviour. Let us write $r \Rightarrow r'$ for a path from r to r' consisting of an arbitrary number (≥ 0) of τ-edges. Then its definition can be rephrased as:

DEFINITION 1.2 Two graphs g and h are *observation equivalent* if there exists a symmetric relation $R \subseteq \text{nodes}(g) \times \text{nodes}(h) \cup \text{nodes}(h) \times \text{nodes}(g)$ (called a *τ-bisimulation*) such that:

1. The roots are related by R.
2. If $R(r,s)$ and $r \to^a r'$ $(a \in A \cup \{\tau\})$, then either $a=\tau$ and $R(r',s)$, or there exists a path $s \Rightarrow s_1 \to^a s_2 \Rightarrow s'$ such that $R(r',s')$.

Furthermore, g and h are *observation congruent* if we also have that

3. (root condition) Root nodes are related with root nodes only.

The root condition was first formulated by BERGSTRA & KLOP [1], and serves to turn the notion of observation equivalence into a congruence with respect to the operators + and ·. It can be proved that observation equivalence and observation congruence are equivalence relations on *G*, and that the latter is the coarsest congruence contained in the former (cf. [5,1,3]). It was shown in [1] that with respect to closed terms the model $G/{\approx^c}$ is completely axiomatized by the theory

$x + y = y + x$	A1	$x\tau = x$	T1
$x + (y + z) = (x + y) + z$	A2	$\tau x = \tau x + x$	T2
$x + x = x$	A3	$a(\tau x + y) = a(\tau x + y) + ax$	T3
$x(yz) = (xy)z$	A4		
$(x + y)z = xy + xz$	A5	$(a \in A \cup \{\tau\})$	

The τ-laws T1-T3 originate from MILNER [5], who gave a complete axiomatization for a similar model with prefixing instead of general sequential composition. From these axioms, it is easy to show why the notion of observation congruence is not preserved under refinement of actions: replacing the action a by the term bc, we obtain $bc(\tau x + y) = bc(\tau x + y) + bcx$ from T3, which obviously is not valid in $G/{\approx^c}$. By T3, we *do* find $bc(\tau x + y) = b(c(\tau x + y) + cx)$ which unfortunately denotes a different process.
Apart from the problem with refinement, it was observed in VAN GLABBEEK & WEIJLAND [3] that observation equivalence does not strictly preserve the branching structure of processes. This is because an important feature of a bisimulation (cf. PARK [6]) is missing for τ-bisimulation, which is the property that any computation in the one process corresponds to a computation in the other, in such a way that all intermediate states of these computations correspond as well. However, in observation congruence, when satisfying the second requirement of definition 1.2 one may execute arbitrarily many τ-steps in a graph without worrying about the status of the nodes that are passed in the meantime.
In order to overcome this problem, in [3] a different notion was introduced, which yields a finer equivalence on graphs.

DEFINITION 1.3 Two graphs g and h are *branching equivalent* if there exists a symmetric relation $R \subseteq \text{nodes}(g)\text{x}\text{nodes}(h) \cup \text{node}(h)\text{x}\text{nodes}(g)$ (called a *branching bisimulation*) such that:

1. The roots are related by R
2. If R(r,s) and $r \to^a r'$ $(a \in A \cup \{\tau\})$, then either $a=\tau$ and R(r',s), or there exists a path of the form $s \Rightarrow s_1 \to^a s'$ such that $R(r,s_1)$ and R(r',s').

Furthermore, g and h are *branching congruent* if we also have that

3. (root condition) Root nodes are related with root nodes only.

Let us write R: g $\leftrightarrow_b$ h if R is a branching bisimulation between g and h and R: g $\leftrightarrow_{rb}$ h if, in addition, R satisfies the root condition. One can prove that the same equivalence is defined when in definition 1.3 *all* intermediate nodes in $s \Rightarrow s_1$ are required to be related with r. Furthermore, observe that a branching bisimulation can also be defined as in definition 1.2, with as extra requirements that $R(r,s_1)$ and $R(r',s_2)$.

It can be proved that branching equivalence and branching congruence are equivalence relations on G. Furthermore, the latter is the coarsest congruence contained in the former. It was shown in [3] that with respect to closed terms, the model $G/\leftrightarrow_{rb}$ is completely axiomatized by the axioms A1-A5 together with

$$x\tau = x \qquad \text{B1}$$

$$x(\tau(y + z) + y) = x(y + z) \qquad \text{B2.}$$

Note that the axioms B1-B2 when applied from left to right only eliminate occurrences of τ's. Using this property, it can be shown that the associated term rewriting system on $G/\equiv_{A1\text{-}A5}$, i.e. G modulo equality induced by the axioms A1-A5, is confluent and terminating. So any two closed branching congruent terms can be reduced to the same normal form.

2. REFINEMENT

In this section we will prove that branching congruence *is* preserved under refinement of actions, and so it allows us to look at actions as abstractions of much larger structures. Consider the following definitions.

DEFINITION 2.1 (substitution)

Let $r: A \rightarrow G$ be a mapping from observable actions to graphs, and suppose $g \in G$. Then, the graph r(g) can be found as follows.

For every edge $r \rightarrow^a r'$ $(a \in A)$ in g, take a copy $\underline{r(a)}$ of r(a) $(\in G)$. Next, identify r with the root node of $\underline{r(a)}$, and r' with all endnodes of $\underline{r(a)}$, and remove the edge $r \rightarrow^a r'$.

Note that in this definition it is never needed to identify r and r', since graphs from G are non-trivial. This way, the mapping r is defined on the domain G. Note that since $\tau \notin A$, τ-edges cannot be substituted by graphs. Finally, observe that every node in g is a node in r(g).

DEFINITION 2.2 (preservation under refinement of actions)

An equivalence $\approx$ on G is said to be *preserved under refinement of actions* if for every mapping $r: A \rightarrow G$, we have: $g \approx h \Rightarrow r(g) \approx r(h)$.

In other words, an equivalence $=$ is preserved under refinement if it is a congruence with respect to every substitution operator r.

Starting from a relation R: $g \leftrightarrow_{rb} h$, we construct a branching bisimulation relation r(R): $r(g) \leftrightarrow_{rb} r(h)$, proving that preserving branching congruence, every edge with a label from A can be replaced by a graph.

DEFINITION 2.3 Let r: $A \rightarrow \mathbf{G}$ be a mapping from observable actions to graphs, $g,h \in \mathbf{G}$ and R: $g \leftrightarrow_{rb} h$. Now r(R) is the smallest relation between nodes of r(g) and r(h), such that:

1. $R \subseteq r(R)$.
2. If $r \rightarrow^{a} r'$ and $s \rightarrow^{a} s'$ $(a \in A)$ are edges in g and h such that R(r,s) and R(r',s'), and both edges are replaced by copies $\underline{r(a)}$ and $\overline{r(a)}$ of r(a) respectively, then nodes from $\underline{r(a)}$ and $\overline{r(a)}$ are related by r(R), only if they are copies of the same node in r(a).

Edges $r \rightarrow^{a} r'$ and $s \rightarrow^{a} s'$ $(a \in A)$ such that R(r,s) and R(r',s'), will be called *related* by R, as well as the copies $\underline{r(a)}$ and $\overline{r(a)}$ that are substituted for them. Observe, that on nodes from g and h the relation r(R) is equal to R. Note that if r(R)(r,s), then r is a node in g iff s is a node in h.

THEOREM (refinement)

Branching congruence is preserved under refinement of actions.

PROOF We prove that R: $g \leftrightarrow_{rb} h \Rightarrow$ r(R): $r(g) \leftrightarrow_{rb} r(h)$ by checking the requirements.

1. The root nodes of r(g) and r(h) are related by r(R).
2. Assume r(R)(r,s) and in r(g) there is an edge $r \rightarrow^{a} r'$. Then there are two possibilities (similarly in case $r \rightarrow^{a} r'$ stems from r(h)):

(i) The nodes r and s originate from g and h. Then R(r,s), and by the construction of r(g) we find that either $a=\tau$ and $r \rightarrow^{\tau} r'$ was already an edge in g, or g has an edge $r \rightarrow^{b} r^*$ and $r \rightarrow^{a} r'$ is a copy of an initial edge from r(b). In the first case it follows from R: $g \leftrightarrow_{rb} h$ that either R(r',s) - hence r(R)(r',s) - or in h there is a path $s \Rightarrow s_1 \rightarrow^{\tau} s'$ such that $R(r,s_1)$ and R(r',s'). By definition, the same path also exists in r(h), and we have $r(R)(r,s_1)$ and r(R)(r',s'). In the second case there must be a path $s \Rightarrow s_1 \rightarrow^{b} s^*$ in h such that $R(r,s_1)$ and $R(r^*,s^*)$. Then, in r(h) we find a path $s \Rightarrow s_1 \rightarrow^{a} s'$ (by replacing $\rightarrow^{b}$ by r(b)) such that $r(R)(r,s_1)$ and r(R)(r',s').

(ii) The nodes r and s originate from related copies $\underline{r(b)}$ and $\overline{r(b)}$ of a substituted graph r(b) (for some $b \in A$), and are no copies of root or endnodes in r(b). Then $r \rightarrow^{a} r'$ is an edge in $\underline{r(b)}$. From r(R)(r,s) we find that r and s are copies of the same

node from r(b). So, there is an edge $s \to^a s'$ in $\overline{r(b)}$ where s' is a copy of the node in r(b), corresponding with r'. Clearly r(R)(r',s').

3. Since for nodes from g and h we have r(R)(r,s) iff R(r,s), the root condition is satisfied. □

With respect to closed terms, the refinement theorem can be proved much easier by syntactic analysis of proofs, instead of working with equivalences between graphs. For observe that the axioms A1-A5 + B1-B2, that form a complete axiomatization of branching congruence for closed terms, do *not* contain any occurrences of (atomic) actions from A. Now assume we have a proof of some equality s=t between closed terms, then this proof consists of a sequence of applications of axioms from A1-A5 + B1-B2. Since all these axioms are universal equations without actions from A, the actions from s and t can be replaced by general variables, and the proof will still hold. Hence, every equation is an instance of a universal equation *without* any actions. Immediately we find that we can substitute arbitrary closed terms for these variables, obtaining refinement for closed terms.

Nevertheless, the semantic proof of the refinement theorem is important as one may wish to generalize the result to models of larger graphs than just finite ones from *G*.

REFERENCES

[1] J.A.BERGSTRA & J.W.KLOP, *Algebra of communicating processes with abstraction*, TCS 37 (1), pp.77-121, 1985.

[2] L.CASTELLANO, G.DE MICHELIS & L.POMELLO, *Concurrency vs Interleaving: an instructive example*, Bulletin of the EATCS 31, pp.12-15, 1987.

[3] R.J.VAN GLABBEEK & W.P.WEIJLAND, *Branching time and abstraction in bisimulation semantics* (extended abstract), Report CS-R8911, Centrum voor Wiskunde en Informatica, Amsterdam 1989, to appear in: proc. IFIP 11th World Computer Congress, San Francisco 1989.

[4] L.LAMPORT, *On interprocess communication. Part 1: Basic formalism*, Distributed Computing 1 (2), pp.77-85, 1986.

[5] R.MILNER, *A calculus of communicating systems*, Springer LNCS 92, 1980.

[6] D.PARK, *Concurrency and automata on infinite sequences*, proc. 5th GI conf. on Th. Comp. Sci. (P.Deussen ed.), Springer LNCS 104, pp.167-183, 1981.

[7] V.R.PRATT, *Modelling concurrency with partial orders*, International Journal of Parallel Programming 15 (1), pp.33-71, 1986.

Towards a theory of (self) applicative communicating processes: a short note.

Henk Goeman, Leiden

March 1989

1 Introduction

It is possible to introduce calculi of processes which are direct extensions of the λ-calculus with several notions taken from theories of concurrency (Process Algebra, CCS, etc.).

Such calculi combine notions of abstraction and (self) application taken from the λ-calculus with notions of (non)deterministic choice, concurrent and sequential composition, communication, synchronisation, encapsulation and hiding taken from the theory of concurrent processes.

In a recent paper [1] Gérard Boudol introduced such a λ-calculus for concurrent and communicating processes, where application appears as (a special kind of) communication and where the β-reduction rule appears as (a special kind of) a communicative interaction law.

A source of inspiration for his work was the striking, but of course intentional, similarity of terms of the form $\lambda x.P$ representing the abstraction mechanism in the λ-calculus and terms of the form $\alpha x.P$ representing synchronised input in CCS [2].

In this short note we will introduce an even more direct combination of the λ-calculus with concepts from concurrency, where application is not translated to other (new or existing) primitive constructs, but is itself maintained as a primitive construct in the combined calculus.

We will first propose a possible syntax for such a calculus. Then we will define a possible operational semantics for it by means of a labelled transition system. Finally we will give several examples of process terms which may give some flavor of the expressive power of the calculus.

Some useful comments were given by Frits Vaandrager on an early draft for this paper.

2 Syntax

Let $x, y, z, \ldots$ denote arbitrary values from a set of symbols $V = v_1, v_2, \ldots$, let $\lambda, \mu, \alpha, \beta, \ldots$ denote arbitrary port names from a set of symbols $\Pi = \pi_1, \pi_2, \ldots$, and let s denote an arbitrary port renaming, i.e. an element from $[\Pi \to \Pi]$. In the following a finite port renaming may be written explicitly as $\alpha_1 \mapsto \alpha'_1, \alpha_2 \mapsto \alpha'_2, \ldots, \alpha_k \mapsto \alpha'_k$.

Now let $P, Q, R, \ldots$ denote arbitrary process terms from the set Γ of process terms inductively defined with the syntax

$$P, Q, R, \ldots \quad ::=$$

$$x \mid (\lambda x.P) \mid (PQ) \mid (\lambda P) \mid (P+Q) \mid (P|Q) \mid (P;Q) \mid (P\backslash\lambda) \mid (P[s]).$$

$(\lambda x.P)$ is called abstraction or input on port λ,
(PQ) is called application,
(λP) is called output on port λ,
$(P+Q)$ is called choice,
$(P|Q)$ is called parallel composition,
$(P;Q)$ is called sequential composition,
$(P\backslash\lambda)$ is called restriction,
$(P[s])$ is called port renaming.

notation

1. outer parentheses are not written;
2. to avoid an excessive use of parentheses the following operator precedences are assumed:
   abstraction < choice < parallel composition < sequential composition < application < output < restriction < renaming;
3. parentheses are also omitted as usual within a repeated abstraction and within a repeated left associative application;
4. the symbol $\equiv$ denotes syntactical equality.

We have chosen for sequential composition as in ACP and for concurrent composition, restriction and port renaming as in CCS. Of course, other possibilities could have been chosen here as well.

3 Semantics

Let a denote an arbitrary action from the set of actions A, where $A = \{\alpha?P,\ \alpha!P,\ \tau \mid \alpha \in \Pi,\ P \in \Gamma\}$. These actions are used as labels in a labelled transition system in $\Gamma \times A \times \Gamma$, generated by the following rules. The rules use also transitions in $\Gamma \times A \times \{\mathsf{nil}\}$.
Please note that nil does not belong to Γ. It should be considered as just a *virtual process*, only used within the transition rules to facilitate the derivations of the real process transitions.

transition rules

1. $\vdash \quad \alpha x.P \xrightarrow{\alpha?Q} P[x := Q]$
2. $P \xrightarrow{\tau} P' \quad \vdash \quad \alpha x.P \xrightarrow{\tau} \alpha x.P'$
3. $P \xrightarrow{\alpha?Q} P' \quad \vdash \quad PQ \xrightarrow{\tau} P'$
4. $P \xrightarrow{\tau} P' \quad \vdash \quad PQ \xrightarrow{\tau} P'Q$
5. $P \xrightarrow{\tau} P' \quad \vdash \quad QP \xrightarrow{\tau} QP'$
6. $\vdash \quad \alpha P \xrightarrow{\alpha!P} \mathsf{nil}$
7. $P \xrightarrow{\tau} P' \quad \vdash \quad \alpha P \xrightarrow{\tau} \alpha P'$
8. $P \xrightarrow{a} \mathsf{nil} \quad \vdash \quad P+Q \xrightarrow{a} \mathsf{nil}$
9. $P \xrightarrow{a} \mathsf{nil} \quad \vdash \quad Q+P \xrightarrow{a} \mathsf{nil}$
10. $P \xrightarrow{a} P' \quad \vdash \quad P+Q \xrightarrow{a} P'$
11. $P \xrightarrow{a} P' \quad \vdash \quad Q+P \xrightarrow{a} P'$
12. $P \xrightarrow{a} \mathsf{nil} \quad \vdash \quad P|Q \xrightarrow{a} Q$
13. $P \xrightarrow{a} \mathsf{nil} \quad \vdash \quad Q|P \xrightarrow{a} Q$
14. $P \xrightarrow{\alpha?R} P',\ Q \xrightarrow{\alpha!R} \mathsf{nil} \quad \vdash \quad P|Q \xrightarrow{\tau} P'$
15. $P \xrightarrow{\alpha?R} P',\ Q \xrightarrow{\alpha!R} \mathsf{nil} \quad \vdash \quad Q|P \xrightarrow{\tau} P'$
16. $P \xrightarrow{\alpha?R} P',\ Q \xrightarrow{\alpha!R} Q' \quad \vdash \quad P|Q \xrightarrow{\tau} P'|Q'$

17. $P \xrightarrow{\alpha?R} P',\ Q \xrightarrow{\alpha!R} Q' \quad \vdash \quad Q|P \xrightarrow{\tau} Q'|P'$

18. $P \xrightarrow{a} P' \quad \vdash \quad P|Q \xrightarrow{a} P'|Q$

19. $P \xrightarrow{a} P' \quad \vdash \quad Q|P \xrightarrow{a} Q|P'$

20. $P \xrightarrow{a} \mathsf{nil} \quad \vdash \quad P;Q \xrightarrow{a} Q$

21. $P \xrightarrow{a} P' \quad \vdash \quad P;Q \xrightarrow{a} P';Q$

22. $P \xrightarrow{\tau} P' \quad \vdash \quad Q;P \xrightarrow{\tau} Q;P'$

23. $P \xrightarrow{\alpha!Q} \mathsf{nil} \quad \vdash \quad P\backslash\beta \xrightarrow{\alpha!Q} \mathsf{nil} \qquad (\alpha \neq \beta)$

24. $P \xrightarrow{\tau} P' \quad \vdash \quad P\backslash\beta \xrightarrow{\tau} P'\backslash\beta$

25. $P \xrightarrow{\alpha?Q} P' \quad \vdash \quad P\backslash\beta \xrightarrow{\alpha?Q} P'\backslash\beta \qquad (\alpha \neq \beta)$

26. $P \xrightarrow{\alpha!Q} P' \quad \vdash \quad P\backslash\beta \xrightarrow{\alpha!Q} P'\backslash\beta \qquad (\alpha \neq \beta)$

27. $P \xrightarrow{\alpha!Q} \mathsf{nil} \quad \vdash \quad P[s] \xrightarrow{s(\alpha)!Q} \mathsf{nil}$

28. $P \xrightarrow{\tau} P' \quad \vdash \quad P[s] \xrightarrow{\tau} P'[s]$

29. $P \xrightarrow{\alpha?Q} P' \quad \vdash \quad P[s] \xrightarrow{s(\alpha)?Q} P'[s]$

30. $P \xrightarrow{\alpha!Q} P' \quad \vdash \quad P[s] \xrightarrow{s(\alpha)!Q} P'[s]$

We will omit here a definition of the substitution construct $P[x := Q]$. This construct should be defined in the usual way where clashes of free and bound occurrences of variables are avoided by means of a suitable renaming of bound variables (α-reduction).

Of course, several rules above may have quite different alternatives, with very natural motivations. It all depends on the desired algebraic properties one may want for the operations on the set of process terms modulo an appropriate observational equivalence, much like the one defined in Boudol's paper [1]. Such an equivalence relation justifies also the equality symbol as used in the next section.

Note especially how the application of a process term P to a process term Q in the proposal above has the effect of making process P behave as a scheduler: it sends process Q to any of its parallel subterms which has evolved to a term with a weak head normal form.
Process application is thus indeed a generalisation of function application!

4 Examples of process terms

1. Let $D \equiv \mu z.\alpha x.\beta(Qx); zz$
   and $O \equiv DD = \alpha x.\beta(Qx); O$.
   The process O represents an object:
   it answers QR on port β for any request R on port α.

2. Let $D \equiv \mu z.(\beta\bot + \alpha x.\beta x); zz$
   and $K \equiv DD = (\beta\bot + \alpha x.\beta x); K$.
   The process K represents a channel with default output $\bot$.

3. Let $D \equiv \mu z.\lambda y.\beta y; zzy + \alpha x.zzx$
   and $R \equiv DD = \lambda y.\beta y; Ry + \alpha x.Rx$
   then $RP = \beta P; RP + \alpha x.Rx$.
   The process RP represents a register with initial content P.
   Note that $\alpha x.Rx$ represents a register without initial content.

4. Let $D \equiv \mu z.\alpha x.\beta x + zz; \beta x$
   and $S \equiv DD = \alpha x.\beta x + S; \beta x$.
   The process S represents a stack to be pushed on port α,
   popped on port β.

5. Let $C \equiv \lambda x.\lambda y.(\, x[\alpha \mapsto \gamma] \mid y[\beta \mapsto \gamma]\,)\backslash\gamma$.
   The process C represents a chaining operator.

Such terms can be used in a very useful way as component process terms in a parallel composition.

Note especially how the usual dichotomy between data objects and program structures is totally absent in our combined calculus: data objects are themselves just component processes.

References

[1] Gérard Boudol, *Towards a Lambda-Calculus for Concurrent and Communicating Systems.*
In: Proc TAPSOFT'89, vol 1: CAAP (J.Díaz,F.Orejas(Eds)), pp 149-161, LNCS 351 (1989).

[2] Robin Milner, *A Calculus of Communicating Systems.*
LNCS 92 (1980).

■

MFCS greetings to Jaco W. de Bakker

This is to acknowledge important contributions of Jaco W. de Bakker to MFCS meetings and to wish him all the best for the future.

Jaco had invited talks at MFCS'76 and MFCS'79, one submitted paper at MFCS'77, and he has been on the program committees for MFCS'78, MFCS'80, MFCS'86, and MFCS'89.

The first appearence of Jaco's name in MFCS environment goes back to MFCS'73 to the citation of his 3 papers in the de Roever's paper, who has also thanked in his paper "J.W. de Bakker for his continuous help,advice and criticism ". The last appearence is so far in at least one of the papers yeasterday accepted for MFCS'89.

The photo shows Jaco lecturing at MFCS'77 in Tatranska Lomnica in a wine cellar. Unfortunately the picture does not show clearly enough that all men in the audience were concentrating very much on the lecture in spite of the fact they had atttractive pictures in the background to investigate.

All these women in the picture are in no way to remind Jaco the past, and also not a way to indicate that there are also other attractive things in the world besides designing concurrency semantic, and also not a way to suggest what to concentrate research on for the next 25 years. The focus and the background of the picture seem to be connected only by the formal semantics.

With many thanks and best wishes

Jozef Gruska

Bratislava
March 18, 1989

The induction rule of De Bakker and Scott

Wim H. Hesselink, February 1989

On the occasion of the Symposium:
"J. W. de Bakker: 25 years of semantics"

Manna's book [M] contains a theorem that goes back to [BS], an unpublished paper of De Bakker and Scott of 1969. This theorem is thus 20 years old. It is called "stepwise computational induction" ([M] 5.5). Disguised as Scott's induction rule, the theorem can also be found in De Bakker's book [B]. More precisely, the book [B] contains a deterministic version (5.37) and a nondeterministic one (7.16). The latter version heavily relies on continuity with respect to the so-called Egli–Milner ordering. Continuity is roughly the same as finite nondeterminacy, for it implies that every necessarily terminating command has a finite set of potential results.

In this note in honour of De Bakker, I would like to announce a generalisation of his induction rule to cases where infinite nondeterminacy is allowed. Actually, a full generalisation is not possible, for I will show a case with infinite nondeterminacy, where the induction rule is not valid. For the proof of the result and for more details, I refer to [H2] and [H3]. The note [H1] contains a small application.

The rule is stated here in a form that differs considerably from the forms in [M] and [B]. The reason is that I re-invented the wheel, so that I also invented my own formalisms. On the other hand, the formalism to be described is more convenient for the applications that I had in mind. In fact, my aim was program transformation rather than correctness. It was only recently, that C. Hemerik pointed out to me that my result was a version of the induction rule of De Bakker and Scott.

Command algebras

Command algebras are introduced to serve as an abstract syntax with a more flexible concept of equality. They are inspired by De Bakker's treatment of nondeterminacy in [B] chapter 7, and also by the process algebras of Bergstra and Klop [BK].

A command algebra A is defined to be a set with constants *fail* $\in A$, *skip* $\in A$ and with binary operators ";" and "$[\!]$" such that the following axioms are satisfied

$$
\begin{array}{lll}
(0) & a \,[\!]\, a = a & a \,[\!]\, b = b \,[\!]\, a \,, \\
 & (a \,[\!]\, b) \,[\!]\, c = a \,[\!]\, (b \,[\!]\, c) & a \,[\!]\, \mathit{fail} = a \,, \\
 & \mathit{fail} \,;\, a = \mathit{fail} & (a \,;\, b) \,;\, c = a \,;\, (b \,;\, c) \,, \\
 & \mathit{skip} \,;\, a = a & a \,;\, \mathit{skip} = a \,, \\
 & (a \,[\!]\, b) \,;\, c = a \,;\, c \,[\!]\, b \,;\, c & a \,;\, (b \,[\!]\, c) = a \,;\, b \,[\!]\, a \,;\, c \,.
\end{array}
$$

A command algebra A is equipped with a partial order "$\leq$" given by

$$a \leq b \;\equiv\; a = a \,[\!]\, b.$$

It turns out that $a \,[\!]\, b$ is the greatest lower bound of a and b with respect to the order. Therefore, it is natural to use the symbol $[\!]$ for arbitrary greatest lower bounds in A. So, if $E \subset A$ has a greatest lower bound in A, then that bound is denoted by $(\,[\!]\, x \in E :: x)$. The algebra A is called *complete* if and only if every subset of A has a greatest lower bound. It turns out that a command $(\,[\!]\, x \in E :: x)$ may be regarded as a nondeterminate choice between the commands $x \in E$.

Now we assume that a command algebra B is given. We regard the elements of B as straight-line commands. We assume that the semantics of B is given by relational semantics. So let Σ_0 be the state space and let $\Sigma = \Sigma_0 \cup \{\bot\}$. The meaning of a command c is given as a subset $M.c$ of the cartesian product $\Sigma_0 \times \Sigma$. A pair (σ, τ) belongs to $M.c$ iff τ is a potential result when command c is called in state σ. A pair $(\sigma, \bot)$ belongs to $M.c$ iff command c need not terminate when called in σ. For a boolean function b on Σ_0, let $?b \in B$ be the command such that $M.(?b)$ consists of all pairs (σ, σ) such that $b.\sigma$ holds, cf. [B] definition 7.8.

Procedures and recursion are treated as follows. We introduce a set H of the occurring procedure names. We then form the polynomial algebra $B[H]$, which consists of the command algebra expressions in elements of B and H modulo the equalities induced by the axioms (0) and the identity relations of B and H, see [H2] section 3.1. The next step is to construct an embedding of algebra $B[H]$ into a complete command algebra $B[H]^*$, see [H3]. This completion satisfies the strong distributive law

$$(\,[\!]\, p \in E, q \in F :: p\,;\,q) = (\,[\!]\, p \in E :: p)\,;\,(\,[\!]\, q \in F :: q)$$

for any pair of nonempty subsets E and F of $B[H]^*$.

A declaration of the procedures is a function $d : H \rightarrow B[H]^*$, where the body of procedure $h \in H$ is defined to be the element $d.h \in B[H]^*$. In this way, recursion and even mutual recursion is possible, and procedure bodies may contain unbounded choice. The semantic function M from $B[H]^*$ to subsets of $\Sigma_0 \times \Sigma$ can be defined as the smallest interpretation with respect to the Egli–Milner ordering and there are equivalent definitions by operational means or by means of predicate transformers, cf. [H0]. In [H2], I use predicate transformer semantics.

The number of recursive procedures need not be finite. In fact, infinite families of procedures are used to allow value parameters and procedure parameters, cf. [H1]. For example, a procedure p with an input parameter v of type V is regarded as a family of commands $p.v$. A call of procedure p with as actual parameter the state function f is defined as the command $p(f) = (\,[\!]\, v \in V :: ?(f = v)\,;\,p.v)$.

The generalised induction rule of De Bakker and Scott

Instead of the generalised correctness formulae as introduced by De Bakker, cf. [B] 5.25 and 7.11, we use congruences, which are defined as follows. A *congruence* on a complete command algebra is defined to be an equivalence relation $\sim$ such that for all commands p, q, r, and s

$$p \sim q \quad \wedge \quad r \sim s \quad \Rightarrow \quad p\,;r \sim q\,;s$$

and that for all sets of commands E, F

$$(\forall p \in E :: (\exists q \in F :: p \sim q)) \quad \wedge \quad (\forall q \in F :: (\exists p \in E :: p \sim q))$$
$$\Rightarrow \quad (\,[\!]\, p \in E :: p) \sim (\,[\!]\, q \in F :: q).$$

Let command $\Omega \in B$ be the abortive command with semantics given by

$$\langle \sigma, \tau \rangle \in M.\Omega \quad \equiv \quad \tau = \bot,$$

and let $da : B[H]^* \to B[H]^*$ be the function such that $da.s$ is obtained from expression s by substituting Ω for every procedure name in expression s. In the same way, we let $d^* : B[H]^* \to B[H]^*$ be the function such that $d^*.s$ is obtained from s by replacing every procedure name h in expression s by its body $d.h$.

In [H3], we introduce a certain subset *Lia* of $B[H]^*$. Let BU be the set of the commands in B that are of finite nondeterminacy. The results of [H3] imply

(1) $\quad B \subset Lia \quad \wedge \quad (BU\,;H\,;B) \subset Lia$
$\quad\quad \wedge \quad (\forall p, q \in Lia :: p \,[\!]\, q \in Lia).$

The generalisation of the induction rule is

Theorem ([H3] 7(14)). Let E be a set of pairs of elements of *Lia* such that

$$(\forall \langle x, y \rangle \in E :: M.(da.x) = M.(da.y)),$$

and that for every congruence $\sim$ on $B[H]^*$ we have

$$(\forall \langle x, y \rangle \in E :: x \sim y) \quad \Rightarrow \quad (\forall \langle x, y \rangle \in E :: d^*.x \sim d^*.y).$$

Then $M.x = M.y$ for all pairs $\langle x, y \rangle \in E$.

In order to show that the condition on *Lia* cannot be omitted, let us consider the following example in which the theorem is not valid. Assume that there is one integer variable i. Let h be the procedure name with the declaration

$$d.h = (?(i > 0)\,;i := i - 1\,;h\,;i := i + 1 \,[\!]\, ?(i \leq 0)).$$

Here, ";" has higher priority than $[\!]$. Clearly, h is semantically equal to *skip*. Let $p \in B$ be a command that is guaranteed to terminate and that assigns to i an arbitrary positive value. Thus, the composition $(p;h)$ is guaranteed to terminate. This implies that

(2) $\quad M.(p;h) \neq M.(p;h \,[\!]\, \Omega).$

On the other hand, let us take E to be the singleton set

$$E = \{\langle (p;h), (p;h \,[\!]\, \Omega) \rangle\}.$$

It is possible to prove (cf. [H2] section 5.6) that both formulae of the theorem are satisfied. By (2), however, the consequent of the theorem is false. So, the condition that the commands be element of *Lia* is violated. The first conclusion is that this nasty condition cannot be omitted. Moreover, from formula (1) we get that $(p;h) \notin Lia$ whereas p and h are both element of *Lia*. Therefore, *Lia* is not closed under composition.

Let me conclude with a more positive remark. The note [H1] contains an application of the theorem to a recursive procedure with an input parameter and a procedural parameter. In this case it is important that the set E of the theorem is allowed to be infinite, and that the commands that occur in E can be complicated expressions.

References

[B] J. de Bakker: Mathematical theory of program correctness. Prentice–Hall International, 1980.

[BS] J.W. de Bakker, D. Scott: A theory of programs. IBM Seminar Vienna, Austria (August 1969). Unpublished notes.

[BK] J.A. Bergstra, J.W. Klop: Algebra of communicating processes, in: J.W. de Bakker, M. Hazewinkel and J.K. Lenstra, eds., Proc. CWI Symp. on Mathematics and Computer Science (North–Holland, Amsterdam, 1986)

[H0] W.H. Hesselink: Interpretations of recursion under unbounded nondeterminacy. Theoretical Computer Science **59** (1988) 211–234.

[H1] W.H. Hesselink: Initialisation with a final value, an exercise in program transformation (WHH 16). To appear in the Proceedings of "Mathematics of program construction", June 1989.

[H2] W.H. Hesselink: Command algebras, recursion and program tranformation (WHH 36). Tech. Rep. CS 8812, Groningen University 1988.

[H3] W.H. Hesselink: Command algebras with unbounded choice (WHH 47). Draft, 1989.

[M] Z. Manna: Mathematical theory of computation. McGraw–Hill Book Company 1974.

Address: Rijksuniversiteit Groningen, Department of Computing Science
P.O. Box 800, 9700 AV Groningen, The Netherlands

Qualitative λ-models as Type Assignment Systems

Furio Honsell - Dip. di Matematica e Informatica - Univ. di Udine (Italy)
Simona Ronchi Della Rocca - Dip. di Informatica - Univ. di Torino (Italy)

Dedicated to J. W. De Bakker in honour of his 25 years of work in Semantics

1. Introduction

Qualitative Domains were introduced by Girard [Gir, 1986] [Gir, 1988], as an alternate means to Scott Domains for providing a semantics for λ-calculus. The main difference between these two notions of domain lies in the way the function space is constructed. The qualitative function space is, in fact, made up only of stable functions, which are a proper subset of the set of all continuous functions. Moreover it is partially ordered with respect to a relation which induces a finer notion of approximation than the one induced by the pointwise ordering. The notion of stable function was first introduced by Berry [Ber, 1978], in the context of sequential functions.

We have studied an alternate description of qualitative λ-models, which allows for a natural definition of a formal system for reasoning about the interpretations of λ-terms. More precisely, we associate, to every qualitative λ-model D, a type assignment system for λ-terms T(D). The set of types of T(D) is isomorphic to the set of atoms of D, and moreover given a λ-term M it is possible to derive a type σ for M if and only if the element of D corresponding to σ belongs to the interpretation of M. A similar connection between Scott's D_∞ -λ-models and a suitable class of type assignment systems for λ-calculus was studied in [Bar, 1983], [Cop, 1984], [Hon, 1984].

In this paper we describe as a type assignment system the standard qualitative λ-model. This particular example can be easily generalized to arbitrary qualitative λ-models.

Describing a qualitative λ-model as a type assignment system gives us the possibility of using standard techniques for studying the fine structure of the model itself. In the particular case studied in this paper, for instance, a normalization property of the type derivation system immediately implies an Approximation Theorem (i.e., the interpretation of a term is the union of the interpretations of its syntactical approximants). Moreover the syntax of the type assignment system that we discuss in this paper provides a deep insight into the structure of qualitative domains, and in our opinion it illuminates the connection between qualitative domains and the coherent semantics for linear logic.

In Section 2 of this paper we define a type assignment system, and we show that it induces a λ-model ***S*** , where the interpretation of a term is the set of types derivable for it. In Section 3 we prove an Approximation Theorem for ***S*** , using a normalization property of the type assignment system. Finally, in Section 4 we define the standard qualitative λ-model ***D*** , with an inverse limit construction, and show that ***S*** and ***D*** are isomorphic.

Throughout the paper we assume the reader familiar with the basic notions and notations of λ-calculus as given in [Bar, 1984]. The definitions of qualitative domains and stable functions are recalled in Section 4.

2. The construction of the model *S*.

Now we introduce a formal system for assigning types to λ-terms and we show that it induces a λ-model.

Let $V=\{\phi_i \mid i\in\omega\}$ be an infinite set of variables. Starting from V, we define two languages, L and L'. Terms of the language L, ranged over by α, are defined as follows:

$$\alpha ::= \phi_1 \mid \phi_2 \mid \ldots \mid [\alpha_1,\ldots,\alpha_n]\rightarrow\alpha \mid [\,]\rightarrow\alpha \qquad (n\geq 1).$$

Terms of L', ranged over by ρ, are defined as follows:

$$\rho := \alpha \mid [\alpha_1,\ldots,\alpha_n] \mid [\,] \qquad (n\geq 1).$$

The intended meaning of $[\alpha_1,\ldots,\alpha_n]$ is the set whose elements are $\alpha_1,\ldots,\alpha_n$, and the intended meaning of $[\,]$ is the empty set. Accordingly, in the sequel, we will take terms of L and L' equivalent up to set-theoretic equality (i.e., reshuffling and repetitions of α_i's in subterms of the form $[\alpha_1,\ldots,\alpha_n]$). Set-theoretic equality is denoted with $=$.

In order to define the set of types, we introduce five predicates whose denotational meaning will be made precise in section 4. The five predicates are:

var $\subset$ L
type $\subset$ L
seq $\subset$ L'
comp $\subset$ L' $\times$ L'
nonc $\subset$ L' $\times$ L'.

These predicates are mutually defined by the rules given in the following definition.

Definition 1.

1) $\dfrac{\mathbf{var}\ \phi}{\mathbf{type}\ \phi}$ 2) $\dfrac{\mathbf{type}\ \alpha \quad \mathbf{seq}\ \rho}{\mathbf{type}\ \rho\rightarrow\alpha}$ 3) $\dfrac{\mathbf{var}\ \phi}{\mathbf{comp}\ \phi\ ,\ \rho}$

4) $_{n\geq 0}$ $\dfrac{(\mathbf{comp}\ \alpha_i,\alpha_j)_{1\leq i,j\leq n}}{\mathbf{seq}\ [\alpha_1,\ldots,\alpha_n]}$ 5) $_{n,m\geq 0}$ $\dfrac{\mathbf{seq}\ [\alpha_1,\ldots,\alpha_n,\alpha'_1,\ldots,\alpha'_m]}{\mathbf{comp}\ [\alpha_1,\ldots,\alpha_n],[\alpha'_1,\ldots,\alpha'_m]}$

6) $\dfrac{}{\mathbf{comp}\ \rho,\rho}$ 7) $\dfrac{\mathbf{comp}\ \rho,\rho'}{\mathbf{comp}\ \rho',\rho}$ 8) $\dfrac{\mathbf{nonc}\ \rho,\rho'}{\mathbf{nonc}\ \rho',\rho}$

9) $\dfrac{\mathbf{comp}\ \rho,\rho' \quad \rho\neq\rho'}{\mathbf{nonc}\ \rho\rightarrow\alpha\ ,\ \rho'\rightarrow\alpha}$ 10) $\dfrac{\mathbf{comp}\ \rho,\rho' \quad \mathbf{nonc}\ \alpha,\alpha'}{\mathbf{nonc}\ \rho\rightarrow\alpha,\rho'\rightarrow\alpha'}$ 11) $\dfrac{\mathbf{comp}\ \rho,\rho' \quad \mathbf{comp}\ \alpha,\alpha' \quad \alpha\neq\alpha'}{\mathbf{comp}\ \rho\rightarrow\alpha\ ,\ \rho'\rightarrow\alpha'}$

12) $\dfrac{\mathbf{nonc}\ \rho,\rho'}{\mathbf{comp}\ \rho\rightarrow\alpha\ ,\ \rho'\rightarrow\alpha}$.

Two remarks, which will be made precise in Section 4, are in order: Rule 9 is distintive of qualitative domains, while Rule 4 is distinctive of binary qualitative domains.

If **type** α holds, then we will say that α is a type. The set of types will be ranged over by σ,τ.
If **seq** ρ holds, then we will say that ρ is a sequence. The set of sequences will be ranged over by γ,δ.

We are now ready to introduce the formal system for assigning types to λ-terms.

Definition 2. i) A **basis** B is a set of pairs $x{:}\sigma$, where σ is a type. dom(B) will denote the

set $\{x \mid x{:}\sigma \in B\}$ and $B_{|E}$ will denote the restriction of B to the set of term variables E.

ii) Types are assigned to terms according to the following three rule schemes, where $B \vdash M{:}\sigma$ denotes that M has type σ under the assumptions recorded in B:

$$(\text{var}) \quad \frac{}{\{x{:}\sigma\} \vdash x{:}\sigma}$$

$$(\to E)_{n\geq 0} \quad \frac{B \vdash M{:}[\sigma_1,\ldots,\sigma_n]\to\sigma \qquad (B_i \vdash N{:}\sigma_i)_{1\leq i\leq n}}{B \cup (\cup_{1\leq i\leq n} B_i) \vdash MN{:}\sigma}$$

$$(\to I)_{n\geq 0} \quad \frac{B \cup \{x{:}\sigma_1,\ldots,x{:}\sigma_n\} \vdash M{:}\sigma \qquad \textbf{seq}\ [\sigma_1,\ldots,\sigma_n] \qquad x\notin dom(B)}{B \vdash \lambda x.M{:}\ [\sigma_1,\ldots,\sigma_n]\to\sigma.}$$

Notice the multiplicative behaviour (in Girard's terminology) of the basis in the rule $(\to E)$.

Definition 3. Two basis B and B' are **coherent** iff $\{x{:}\sigma,\ x{:}\sigma'\} \subset B \cup B' \Rightarrow$ **comp** σ,σ'.

Some structural properties of this type assignment system are given in Lemma 4.

Lemma 4.i) Let $B \vdash M{:}\sigma$. Then $\{x{:}\tau_1,\ldots,x{:}\tau_m\} \subset B$ implies that there are at least m occurrences of x in M.
ii) Let $B \vdash \lambda x.M{:}\sigma$. Then $\sigma = \rho\to\tau$ for some ρ such that **seq** ρ.
iii) Let $B \vdash M{:}\sigma$, $B' \vdash M{:}\tau$ and let B and B' be coherent; then **comp** σ,τ.
iv) Let $B \vdash M{:}\sigma$, $B' \vdash M{:}\tau$, let B and B' be coherent and let $\sigma=\tau$; then $B=B'$.

Proof.i) and ii) are immediate.
iii) and iv) will be proved simultaneously by induction on M.
$M \equiv x$. Obvious.
$M \equiv PQ$. $B \vdash M{:}\sigma$ and $B' \vdash M{:}\tau$ implies that there are two derivations of the following shape:

$$(\to E) \frac{B_0 \vdash P{:}[\sigma_1,\ldots,\sigma_n]\to\sigma \quad (B_i \vdash Q{:}\sigma_i)_{1\leq i\leq n}}{B = B_0 \cup (\cup_{1\leq i\leq n} B_i) \vdash PQ{:}\sigma} \qquad (\to E) \frac{B'_0 \vdash P{:}[\tau_1,\ldots,\tau_m]\to\tau \quad (B'_i \vdash Q{:}\tau_i)_{1\leq i\leq m}}{B' = B'_0 \cup (\cup_{1\leq i\leq m} B'_i) \vdash PQ{:}\tau}$$

Since two subsets of two coherent basis are in turn coherent , by induction **comp** $[\sigma_1,\ldots,\sigma_n]\to\sigma,\ [\tau_1,\ldots,\tau_m]\to\tau$ and **comp** $(\sigma_i,\tau_j)_{1\leq i\leq n, 1\leq j\leq m}$, which implies **comp** σ,τ. Moreover, if $\sigma=\tau$ then $[\sigma_1,\ldots,\sigma_n] = [\tau_1,\ldots,\tau_n]$ and iv) follows directly by induction.
$M \equiv \lambda x.P$. If $B \vdash M{:}\sigma$ and $B' \vdash M{:}\tau$ then, by ii) we must have $\sigma = [\sigma_1,\ldots,\sigma_n]\to\sigma'$ and $\tau = [\tau_1,\ldots,\tau_m]\to\tau'$, with $m,n\geq 0$. This implies that there are two derivations of the following

shape:

$$(\to I)\ \frac{B\cup\{x:\sigma_1,\ldots,x:\sigma_n\}\vdash P:\sigma' \quad \mathbf{seq}\ [\sigma_1,\ldots,\sigma_n] \quad x\notin dom(B)}{B\vdash \lambda x.P: [\sigma_1,\ldots,\sigma_n]\to\sigma'}$$

and

$$(\to I)\ \frac{B'\cup\{x:\tau_1,\ldots,x:\tau_m\}\vdash P:\tau' \quad \mathbf{seq}\ [\tau_1,\ldots,\tau_m] \quad x\notin dom(B')}{B'\vdash \lambda x.P: [\tau_1,\ldots,\tau_m]\to\tau'.}$$

If **comp** $[\sigma_1,\ldots,\sigma_n]$, $[\tau_1,\ldots,\tau_m]$, then by induction **comp** σ',τ'. If $\sigma'\neq\tau'$ then, by definition of **comp**, this implies **comp** σ,τ, else, if $\sigma'=\tau'$,by induction on iv), $[\sigma_1,\ldots,\sigma_n]=[\tau_1,\ldots,\tau_m]$, and so $\sigma=\tau$. If **nonc** $[\sigma_1,\ldots,\sigma_n]$, $[\tau_1,\ldots,\tau_m]$, iii) follows by definition of **comp**. iv) is immediate by induction.

□

Now we are able to define the model $\mathcal{S}$.

Definition 5. $\mathcal{S} \equiv (S,\circ,[\![\ ]\!])$, where:
$S \equiv <\{A\subset L \mid \forall\alpha_1,\alpha_2 \in A.\ \mathbf{comp}\ \alpha_1,\alpha_2\}, \subset >$.
$s_1 \circ s_2 = \{\alpha \mid [\alpha_1,\ldots,\alpha_n]\to \alpha \in s_1,\ \{\alpha_1,\ldots,\alpha_n\} \subset s_2\}$ $\quad (s_1,s_2 \subset S)$.
Given an environment ξ:Var→S, where Var is the set of term variables, ξ induces a basis $B_\xi = \{x:\alpha \mid \alpha\in \xi(x)\}$; then $[\![M]\!]_\xi = \{\sigma \mid \exists B.\ B\vdash M:\sigma$ and $B\subset B_\xi\}$.

Theorem 6. $\mathcal{S}$ is a λ-model.
Proof. First of all, it is necessary to prove that $\forall M\in\Lambda.\ \forall\xi.[\![M]\!]_\xi \in S$. This is an immediate consequence of Lemma 4.iii).
Then $\mathcal{S}$ can be proved to be a λ-model by showing that it satisfies the six conditions defining a λ-model given by Hindley and Longo in [Hin, 1980], i.e.:
1) $\quad [\![x]\!]_\xi = \xi(x)$;
2) $\quad [\![MN]\!]_\xi = [\![M]\!]_\xi \circ [\![N]\!]_\xi$;
3) $\quad [\![\lambda x.M\,]\!]_\xi \circ s = [\![M]\!]_{\xi[s/x]}$;
4) $\quad (\forall x\in FV(M).\ [\![x]\!]_\xi = [\![x]\!]_{\xi'}) \Rightarrow [\![M]\!]_\xi = [\![M]\!]_{\xi'}$;
5) $\quad [\![\lambda x.M\,]\!]_\xi = [\![\lambda x.M[y/x]\,]\!]_\xi$, if $y \notin FV(M)$;
6) $\quad (\forall s\in S.\ [\![M]\!]_{\xi[s/x]} = [\![N]\!]_{\xi[s/x]}) \Rightarrow [\![\lambda x.M\,]\!]_\xi = [\![\lambda x.N\,]\!]_\xi$.
This will be proved by induction on terms.
1) $[\![x]\!]_\xi = \{\sigma \mid \exists B.\ B\vdash x:\sigma$ and $B\subset B_\xi\} = \{\sigma \mid \sigma\in\xi(x)\}$;
2) $[\![MN]\!]_\xi = \{\sigma \mid \exists B.\ B\vdash MN:\sigma$ and $B\subset B_\xi\} =$
$\{\sigma \mid \exists B,B_1,\ldots,B_n\subset B_\xi.\ B\vdash M:[\sigma_1,\ldots,\sigma_n]\to\sigma$ and $(B_i\vdash N:\sigma_i)_{i\le n}\}$ = (by induction)
$\{\sigma \mid [\sigma_1,\ldots,\sigma_n]\to\sigma\in [\![M]\!]_\xi$ and $(\sigma_i\in [\![N]\!]_\xi)_{i\le n}\} = [\![M]\!]_\xi \circ [\![N]\!]_\xi$ (by def. of $\circ$);
3) $[\![\lambda x.M\,]\!]_\xi \circ s = \{\sigma \mid \exists B.\ B\subset B_\xi$ and $B\vdash\lambda x.M: [\sigma_1,\ldots,\sigma_n]\to\sigma$ and $(\sigma_i\in s)_{i\le n}\} =$
$\{\sigma \mid \exists B\subset B_\xi\ B\cup\{x:\sigma_1,\ldots,x:\sigma_n\}\vdash M:\sigma$ and $(\sigma_i\in s)_{i\le n}\} =$
$\{\sigma \mid \exists B'\subset B_{\xi[s/x]}.B'\vdash M:\sigma\}$;
4) $(\forall x\in FV(M).\ [\![x]\!]_\xi = [\![x]\!]_{\xi'}) \Rightarrow (\forall x\in FV(M).\{\sigma \mid \sigma\in\xi(x)\} = \{\sigma \mid \sigma\in\xi'(x)\}) \Rightarrow$

$(B_{\xi|FV(M)} = B_{\xi'|FV(M)} \Rightarrow$ (by Lemma 4.i) $\{\sigma \mid \exists B.\ B \vdash M:\sigma$ and $B \subset B_\xi\} =$
$\{\sigma \mid \exists B.\ B \vdash M:\sigma$ and $B \subset B_{\xi'}\} \Rightarrow [\![M]\!]_\xi = [\![M]\!]_{\xi'}$;

5) immediate from the definition of $[\![\ ]\!]$;

6) $[\![\lambda x.M]\!]_\xi =$ (by Lemma 4.ii)$\{[\sigma_1,\ldots,\sigma_n]\to\sigma \mid \exists B \subset B_\xi.\ B \vdash \lambda x.M: [\sigma_1,\ldots,\sigma_n]\to\sigma\} =$
$\{[\sigma_1,\ldots,\sigma_n]\to\sigma \mid \exists B \subset B_\xi.B \cup \{x:\sigma_1,\ldots,x:\sigma_n\} \vdash M:\sigma$ and **seq** $[\sigma_1,\ldots,\sigma_n]$ and $x \notin dom(B)\} =$
$\{[\sigma_1,\ldots,\sigma_n]\to\sigma \mid \exists B' \subset B_{\xi[[\sigma_1,\ldots,\sigma_n]/x]}.\ B' \vdash M:\sigma\} =$ (since $\forall s \in S.\ [\![M]\!]_{\xi[s/x]} =$
$[\![N]\!]_{\xi[s/x]}$) $\{[\sigma_1,\ldots,\sigma_n]\to\sigma \mid \exists B \subset B_{\xi[[\sigma_1,\ldots,\sigma_n]/x]}.\ B' \vdash N:\sigma\} = [\![\lambda x.N]\!]_\xi$. □

3. The Approximation Theorem.

Every derivation D of $B \vdash M:\sigma$ in the above type assignment system is normalizable. Here normalizable means that D can be transformed into a derivation D' of $B \vdash M':\sigma$ where no application of the rule ($\to$I) in D' is immediately followed by an application of the rule ($\to$E) and M' is a β-reduct of M. Using this fact we will show that the interpretation of a term in ***S*** is the collection of the interpretations of its syntactical approximants. This will be called the Approximation Theorem for the model ***S***.

Definition 7. i) Let D be the deduction: $B \vdash M:\sigma$. A **cut** in D is an application of the rule ($\to$I) immediately followed by an application of the rule ($\to$E);
ii) The **degree** of a cut is the number of type symbols occurring in the premises of the application of the rule ($\to$E) determining the cut.
iii) The **degree** of a deduction D, G(D), is the pair $<d,n>$ where n is the number of cuts in D and d is the maximum degree of all cuts in D.
iv) A deduction D is **normal** iff $G(D) = <0,0>$.

We consider the pairs ordered in lexicographic order (i.e., $<d,n> \sqsubseteq <d',n'>$ iff ($d \sqsubseteq d'$) or ($d=d'$ and $n \sqsubseteq n'$)).

Lemma 8. $D:B \vdash M:\sigma$ and $G(D) > 0$ implies that there exists D' such that $D' \vdash M':\sigma$, where M β-reduces to M' and $G(D') < G(D)$.

Proof. We have to distinguish two cases, according to the number of premises of the ($\to$E) rule which determines the cut.

1) At least one of the cuts with the maximum degree in D is of the following shape:

$$\dfrac{\dfrac{B \vdash P:\tau \quad x \notin dom(B)}{B \vdash \lambda x.P:[\ ]\to\tau}(\to I)}{B \vdash (\lambda x.P)Q:\tau.}(\to E)$$

This implies that D: $B \cup B' \vdash C[(\lambda x.P)Q]:\sigma$, for a suitable context C[ ].
Then, if x occurs in P at all, x occurs in subterms of P $S_i[x]$ ($i \geq 0$), which occur in subderivations of D of the shape:

$$D_i: \quad \dfrac{B_i' \vdash R:[\ ]\to\sigma_i}{B_i' \vdash RS_i[x]:\sigma_i.}(\to E)$$

Then D' is obtained from D by performing the following three operations:

i) replacing every D_i with:

$$\frac{D_i' \quad B_i' \vdash R:[\,]\to\sigma_i}{B_i' \vdash RS_i\,[Q\backslash x]:\sigma_i}\ (\to E)$$

ii) replacing $(\lambda x.P)Q$ and every descendent of it with $P[Q\backslash x]$
iii) deleting the cut.
Thus we have D': $B\cup B' \vdash C[P[Q\backslash x]]:\sigma$ and $G(D') < G(D)$.
2) All the cuts with the maximum degree in D are of the following shape:

$$\frac{\dfrac{B\cup\{x:\sigma_1,\dots,x:\sigma_n\}\vdash P:\sigma \quad \mathbf{seq}\ [\sigma_1,\dots,\sigma_n] \quad x\notin dom(B)}{B\vdash \lambda x.P:[\sigma_1,\dots,\sigma_n]\to\sigma}\ (\to I) \qquad (B_i\vdash Q:\sigma_i)_{i\leq n}}{B\cup(\cup_{i\leq n}B_i)\vdash(\lambda x.P)Q:\sigma.}\ (\to E)$$

Pick one of these. This implies that D: $B\cup B' \vdash C[(\lambda x.P)Q]:\sigma$, for a suitable context C[ ]. Now, there are $m\geq n$ occurrences of x in P, by Lemma 4.i). Exactly n of these occurrences occur in subderivations of D_i $(1\leq i\leq n)$, consisting of an application of the (var) rule:

$$D_i:\quad \frac{}{x:\sigma_i\vdash x:\sigma_i}\ (var).$$

The remaining m-n occurrences of x are in subterms of P for which no type has been derived in D. Then D' is obtained from D by performing the following four operations:
i) replacing every D_i with D_i' :$B_i\vdash Q:\sigma_i$
ii) handling the remaining m-n occurrences of x for which no type has been derived in D as in 1)
iii) replacing $(\lambda x.P)Q$ and every descendent of it with $P[Q\backslash x]$
iv) deleting the cut.
Thus we have D': $B\cup B' \vdash C[P[Q\backslash x]]:\sigma$ and $G(D') < G(D)$.

□

The following theorem is an easy consequence of the lemma we have just proved.

Theorem 9. If $D:B\vdash M:\sigma$ then there exists a normal derivation D' and a term M' such that M β-reduces to M'and $D':B\vdash M':\sigma$.

We will now recall the notion of approximate normal form first introduced in [Wad,1978] in order to discuss the interpretation of non-terminating λ-terms.

Definition 10. i) The set $\mathcal{A}$ of the **approximate normal forms** is defined inductivily as:
- a term variable belongs to $\mathcal{A}$;
- the constant Ω belongs to $\mathcal{A}$;
- if $A_1,\dots,A_n$ belong to $\mathcal{A}$, then $\lambda x_1,\dots,x_m.zA_1\dots A_n$ belongs to $\mathcal{A}$, for any term variables $x_1,\dots,x_m,z$.
ii) If $M\in\Lambda$, the set of the **approximants** of M is:
$\mathcal{A}(M)=\{A\in \mathcal{A} \mid \exists M'.\ M\ \beta\text{-reduces to } M' \text{ and } A \text{ and } M' \text{ match up to subterms of } M' \text{ corresponding to occurrences of } \Omega \text{ in } A\}$.

Approximation Theorem. $B\vdash M:\sigma$ iff $\exists A\in \mathcal{A}(M).\ B\vdash A:\sigma$.
(i.e., $[\![M]\!]_\xi=\{[\![A]\!]_\xi \mid A\in \mathcal{A}(M)\}$).
Proof. ($\Rightarrow$) $B\vdash M:\sigma$ implies (by Lemma 8) $\exists D:B\vdash M':\sigma$ and M β-reduces to M' and D is normal. Let A be the approximant of M obtained from M'by replacing with Ω every subterm of M' to which no type has been assigned by D. Clearly $B\vdash A:\sigma$. If D has assigned a type to every

subterm of M', then M' is in normal form and A=M'.

($\Leftarrow$) Let A be an approximant of M such that $\exists D:B\vdash A:\sigma$. Then M reduces to M', where M' is obtained from A by replacing the occurrences of Ω with suitable subterms, say $N_1,\dots,N_p$. Every occurrence of Ω in A must occur in a subderivation of the shape:

$$(\rightarrow E)\frac{B'\vdash A':[\,]\rightarrow\tau}{B'\vdash A'\Omega:\tau.}$$

So a derivation $D':B\vdash M':\sigma$ can be obtained from D simply by replacing in D the occurrences of Ω with $N_1,\dots,N_p$ respectively. Since $[\![\ ]\!]$ satisfies β-equality, by Theorem 6, we have that $B\vdash M':\sigma$ implies $B\vdash M:\sigma$.

□

The Approximation Theorem is a powerful tool for investigating the theory induced by a model. In this case it implies immediately, for instance, that the theory of the model ***S*** is sensible, and that $[\![Y]\!]$ is Tarski's least fixed point operator.

Moreover, using the Approximation Theorem and following the argument in [Ron, 1982], one can characterize the theory of ***S*** as: $\forall\xi.\ [\![M]\!]_\xi \sqsubseteq [\![N]\!]_\xi$ if and only if

$\forall C[\,]$.(if C[M] reduces to a head-normal-form with no initial abstractions then the same holds for C[N]).

This is the same theory as the one induced by the filter model [Bar,1983]. Equationally it is the same as the theory of Scott's P_ω.

We will end this section pointing out the following interesting fact:

Multiplicity Theorem. Let $\lambda x.A$ be an approximate normal form. There are exactly n occurrences of the variable x in A if and only if n is the maximum integer such that there is a derivation $D:B\vdash\lambda x.A:[\sigma_1,\dots,\sigma_n]\rightarrow\sigma$.

Proof. The theorem follows immediately noticing that, given an approximate normal form A it is always possible to build a derivation D where every subterm of A has a type, but Ω's .

□

4. Isomorphism Theorem.

In this section we give a semantics to the type assignment system that we introduced in Section 2. More precisely we will show that the model ***S*** is isomorphic to a particular binary qualitative domain ***D*** which is a λ-model.

First let us recall some definitions about qualitative λ- models [Gir, 1986].

Definition 11. i) A **qualitative domain** D is a set of sets such that:
- $\emptyset\in D$
- D is closed under directed unions
- if $a\in D$ and $b\subset a$, then $b\in D$;

ii) The union of the set of the atomic elements of D i.e. $\{ z \mid \{z\}\in D \}$, is denoted with $|D|$;

iii) Let D and D' be two qualitative domains. A function $F: D\rightarrow D'$ is **stable** iff:
- $a\subset b\in D \Rightarrow F(a)\subset F(b)$
- $F(\cup_{i>0} a_i)=\cup_{i>0} F(a_i)$, provided $a_i\subset a_j$ for $i\leq j$
- $a\cup b\in D \Rightarrow F(a\cap b) = F(a)\cap F(b)$;

iv) Let D and D' be two qualitative domain, and let $F: D\rightarrow D'$ be a stable function. The **trace** of F is:

$Tr(F) = \{(a,z) \mid a$ is a finite element of D, $z\in|D'|$, $z\in F(a)$ and $z\notin F(a')$ for all $a'\subset a\}$;

v) A qualitative domain D is a λ-model iff there are two stable functions H and K such that:

$H: D \rightarrow [D \rightarrow_s D]$, $K: [D \rightarrow_s D] \rightarrow D$ and $H \circ K = Id_{[D \rightarrow sD]}$
where $[D \rightarrow_s D]$ denotes the qualitative domain of the traces of the stable functions from D to D, partially ordered by inclusion.

Definition 12. Let D be the qualitative domain defined as the standard inverse limit solution of the following equation:

$$D = P(V) \times [\, D \rightarrow_s D \,]$$

where P(V) denotes the power set of the set V of variables, ordered by inclusion.
Up to isomorphisms, D is defined as:

$D = \lim_{n \geq 0} D_n$, where $D_0 = \{\varnothing\}$, $D_{n+1} = P(V) \times [D_n \rightarrow_s D_n]$.

Isomorphism Theorem. S and D are isomorphic.
Proof (sketch). One can easily verify that S is a qualitative domain; in particular $|S| = \{\alpha \mid$ **type** $\alpha\}$. An isomorphism I between the supports of S and D can be defined in the following way:
$I(\phi) = (\{\phi\}, \varnothing)$
$I([\sigma_1,\ldots,\sigma_n] \rightarrow \sigma) = (\varnothing, F)$, where $Tr(F) = \{(\,\{I(\sigma_1),\ldots, I(\sigma_n)\}, I(\sigma))\}$.
Then I can easily be extended to all points of S and D , since the definition of **comp** is such that
$\{\alpha\} \cup \{\alpha'\} \in S \iff$ **comp** $\alpha, \alpha' \iff \exists d \in D\ (\forall \phi \in \{\alpha\} \cup \{\alpha'\}.\{\phi\} \in \pi_1(\text{unfold}(d))$ and $\forall [\sigma_1,\ldots,\sigma_n] \rightarrow \sigma \in \{\alpha\} \cup \{\alpha'\}.\ (\,\{I(\sigma_1),\ldots, I(\sigma_n)\}, I(\sigma)) \in \pi_2(\text{unfold}(d)))$.

□

We can now show that the five predicates introduced in Section 2 are abstract syntactic counterparts of qualitative-domain-theoretic concepts. Let μ and μ' be elements of S and let I be the isomorphism between S and D . The following relations hold:

var (μ)	$\Leftrightarrow$	$I(\mu)$ is a non–functional atom of D
type (μ)	$\Leftrightarrow$	$I(\mu)$ is an atom of D
seq (μ)	$\Leftrightarrow$	$I(\mu)$ is an element of D
comp (μ, μ') and **type** (μ) and **type** (μ')	$\Leftrightarrow$	$\{I(\mu), I(\mu')\}$ is an element of D
comp (μ, μ') and **seq** (μ) and **seq** (μ')	$\Leftrightarrow$	$I(\mu) \cup I(\mu')$ is an element of D
nonc (μ, μ') and **type** (μ) and **type** (μ')	$\Leftrightarrow$	$\{I(\mu), I(\mu')\}$ is not an element of D
nonc (μ, μ') and **seq** (μ) and **seq** (μ')	$\Leftrightarrow$	$I(\mu) \cup I(\mu')\}$ is not an element of D.

References

[Bar, 1983] Barendregt H., Coppo M., Dezani-Ciancaglini M., *A Filter Lambda Model and the Completeness of Type Assignment*, The Journal of Symbolic Logic, 48,4, pp. 931- 940.

[Bar, 1984] Barendregt H.,*The Lambda Calculus: its Syntax and Semantics*, revised version, North-Holland.

[Ber, 1978] Berry G.,*Stable Models of Typed Lambda Calculi*, LNCS,Springer-Verlag, 62, pp. 72-89.

[Cop, 1984] Coppo M., Dezani-Ciancaglini M., Honsell F., Longo G., *Extended Type Structures and Filter Lambda Models*, Logic Colloquium 82, G.Lolli et al (eds), Elsevier Science Publishers B.V., North Holland, pp. 241-262.

[Gir, 1986] Girard J.Y., *The System F of Variable Types, fifteen years later*, Theoretical Computer Science, 45, pp. 159-192.

[Gir, 1988] Girard J.Y.,*Normal Functors, Power Series and Lambda Calculus*, Annals of Pure and Applied Logic, 37, 2, pp. 129-177.

[Hin, 1980] Hindley R., Longo G., *Lambda Calculus Models and Extensionality*, Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik, 26, pp.286-310.

[Hon, 1984] Honsell F., Ronchi Della Rocca S., *An Approximation Theorem for Topological Lambda Models and the Topological Incompleteness of Lambda Calculus*, to appear in Journal of Computer and Systems Sciences.

[Ron,1982] Ronchi Della Rocca S., *Characterization Theorems for a filter lambda model*, Information and Control,54,201-216.

[Wad, 1978] Wadsworth C.P., *Approximate reductions and λ-calculus Models*, SIAM Journal of Computing, 7, 3, pp. 337-356.

A Compositional Semantics for Statecharts*

J. Hooman
S. Ramesh†
W. P. de Roever

Eindhoven University of Technology
Dept. of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven, The Netherlands

Abstract

Statecharts is a behavioral specification language proposed for specifying large real-time, event driven, reactive systems. It is a graphical language based on finite state machines extended with many features like hierarchy, concurrency and broadcast communication. We give a compositional syntax and a denotational semantics for Statecharts.

1 Introduction

This paper concerns the semantics of a specification language for describing real-time reactive systems. Real-time reactive systems usually run forever, interact continuously with their environment, and have critical time requirements. Typical examples are telecommunication networks and avionic systems. Formal description of real-time reactive systems is an important area of research judging by the sheer number of proposed specification languages such as Statecharts [Har87,Har88], Esterel [BC85], Modecharts [JM89], Lustre [BCH85] and Signal [GB86]. All these specification languages are based on operational descriptions that characterize how a system evolves.

The behavioral specification language considered here is Statecharts [Har87]. Realizing the intuitive and pictorial appeal of finite state machines, Statecharts has been designed on the basis of such state machines. But it is free from the limitations of state machines, such as sequentiality, unstructuredness and exponential growth of states when describing concurrency. Indeed, Statecharts are double exponentially more succinct than state machines (Harel). Quoting [Har88],

Statecharts=finite state machines+depth+orthogonality+broadcast communication.

*This work was supported by ESPRIT Project 937: Debugging and Specification of Ada Real-Time Embedded Systems (DESCARTES).

†Supported by the Foundation for Computer Science Research in the Netherlands (NFI) with financial aid from the Netherlands Organization for Scientific Research (NWO).

Depth is achieved in Statecharts by allowing super states containing substates or even complete statecharts. When such a super state is exited all the computations inside are terminated. Super states may consist of orthogonal sub-statecharts which are executed in parallel. Orthogonal components interact with each other and with their environment by means of events which are broadcast throughout the whole system. External events or events generated in one component can cause new events in another component which, in turn, can cause more events. Thus a single event can give rise to a whole chain of events all of which are assumed to take place simultaneously; this assumption is essentially Berry's synchrony hypothesis [BC85] according to which a system is infinitely faster than its environment. It facilitates the specification task by abstracting from internal reaction times.

The synchrony hypothesis might introduce causal paradoxes like an event causing itself. In Esterel [BC85] causal paradoxes are syntactically disallowed whereas in Statecharts causal relationships are respected and paradoxes are removed semantically. Real-time is incorporated in Statecharts by having an implicit clock, by allowing transitions to be triggered by *time-outs* relative to this clock and by requiring that if a transition can be taken then it should be taken immediately.

1.1 Overview of our work

Our aim is to develop a *compositional* denotational semantics for Statecharts. This requires syntactical operators for building large statecharts from smaller ones. Our compositional semantics is based on a syntax for Statecharts which has been proposed in [HGR88]. Section 2.1 first informally introduces Statecharts and Section 2.2 contains this syntax.

The denotations describe observable entities (i.e. the events generated by a statechart), but also denote non-observable entities such as the causal relation between events. These non-observable entities, which can be sensed by a suitable program context, are needed to obtain a compositional semantics. In [HGR88] a compositional semantic model with minimal amount of non-observable entities (i.e. a fully abstract semantics) has been presented. This semantics forms the basis of our axiomatic system. The denotations of this semantics are prefix-closed sets of linear histories; infinite computations are represented by all their finite prefixes. Our semantics forms the basis for a proof system in which Statecharts are related to property based specifications. In order to express liveness properties, we do not use prefix closed sets. Our histories represent complete (possibly infinite) computations. Section 3 contains the details of the modified semantic model.

2 Syntax

2.1 General overview

A statechart can be considered as a tree of states, with the root state as the initial state. The leaves of the tree are basic states, like the states in a finite state machine. Other states

are super states containing their sons (in the tree) as substates. There are two types of super states: AND-states and OR-states. For instance, the statechart in Figure 1 has root

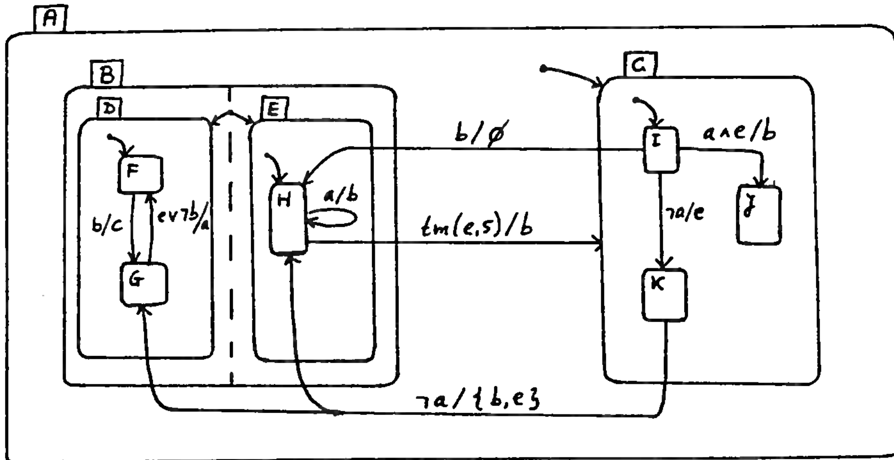


Figure 1:

state A and leaves F, G, H, I, J and K. A is an OR-state, with substates B and C. B is an AND-state (indicated by the dashed line) with D and E as its substates, called orthogonal components, whereas C is an OR-state having I, J and K as its substates. States are entered/exited either explicitly by taking a transition or implicitly because certain other states are entered/exited. Entering an AND-state (OR-state resp.) results in entering *all* (*exactly one* resp.) of its substates implicitly. In Figure 1: entering AND-state B results in entering both D and E, entering OR-state C results in entering exactly one of its substates I, J and K. Similarly, entering an orthogonal component of an AND-state results in implicitly entering all other components of this AND-state. When entering a super state the particular substate(s) which should be entered, is (are) marked by a default arrow, drawn as a transition with no source state, e.g. the default arrow inside C pointing to I.[1] When the transition from I to H is taken, H is entered explicitly and B, D, E, F are entered implicitly. Note that with a forked transition such as the one from K, more than one orthogonal component can be entered explicitly. Transitions between orthogonal components are not allowed (e.g. no transitions between F and H). When a state is exited all its substates are exited implicitly. Exiting an orthogonal component implies an implicit exit of all its orthogonal partners. So the transition from H to C leads to an implicit exit of D (and its substate F or G).

Transitions have labels of the form *'event part/action part'*.[2] The event part is a boolean expression involving atomic events a, b, e ,... - signals without measurable duration. These

[1] Unlike [Har87], we attach a default arrow to every super state; so also to AND-states.

[2] For the sake of simplicity we do not consider the general syntax of labels given in [HPPSS87]. There a

events can be generated by the outside world as an input to the statechart as well as by the statechart itself. The event expression specifies when the transition is enabled. The action part is a set of atomic events which are generated when the transition is taken (a singleton is denoted by its element).

Execution in orthogonal components proceeds concurrently and events generated in one component are broadcast throughout the system, possibly triggering new transitions in other components. This will in general lead to a whole chain of transitions which, by the synchrony hypothesis, take place simultaneously in a single (time) step. The set of transitions taken in a step is a *maximal* set in which there is at most one transition per orthogonal component and there exists a causal relationship between transitions: each transition is enabled by either external events or events generated by other transitions. The general idea is that staying in a state takes some time, whereas taking a transition is instantaneous.
In our example the system can be in the states A, B, D, E, F and H simultaneously. When a is generated externally in this configuration the transition from (and to) state H will generate b, causing a transition from F to G which generates c.

A transitions with event part a is taken when a is generated somewhere in the system. The meaning of $a \wedge b$ (resp. $a \vee b$) is: a transition with this event part is taken in a step if both a and b (resp. a or b) are generated somewhere in the system in this step. λ is a special event which occurs (by definition) in every step. $tm(e, n)$ denotes a *time-out* event which is generated at a particular step if n time steps earlier event e has happened. A transition with event part $\neg a$ is taken in a step if a is not generated at all during this step.[3] In our syntax, negations are immediately succeeded by atomic events, time-outs or λ.

2.2 Syntax of Statecharts

The objects obtained using our syntax are in general unfinished statecharts having arcs without either source or target state. In the sequel we use the word statechart for both unfinished and finished statecharts.
The primitive objects (see Figure 2.2) of our syntax are so called

- Basic Statecharts: $[I, O, S]$, where S is a state name, I a set of incoming arcs and O a set of outgoing arcs. Only the outgoing arcs are labeled with an event/action pair.

We have the following operators (let B be a basic statechart, U, U_1, U_2 be statecharts, T, T_1, T_2 be transition names and let a be the name of an atomic event):

label includes an additional condition part, variable assignments are allowed in action parts and there are special events to signal entry and exit of a state. The proposed axiomatic system can be easily extended to the general case.

[3]There are several possible interpretation of $\neg a$, here we take the approach recently advocated by Pnueli [PS88].

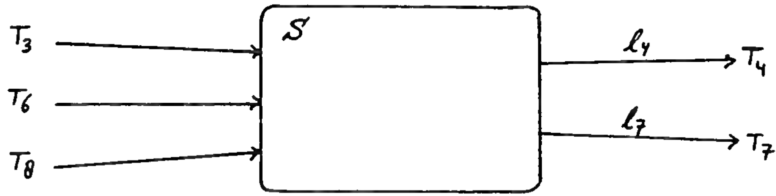


Figure 2: Basic Statechart $[I, O, S]$, with $I = \{T_3, T_6, T_8\}$ and $O = \{T_4, T_7\}$

- Statification: $Stat(B, U, T)$; makes (the state of) B a super state with U inside it and the incoming transition T of U as its default.

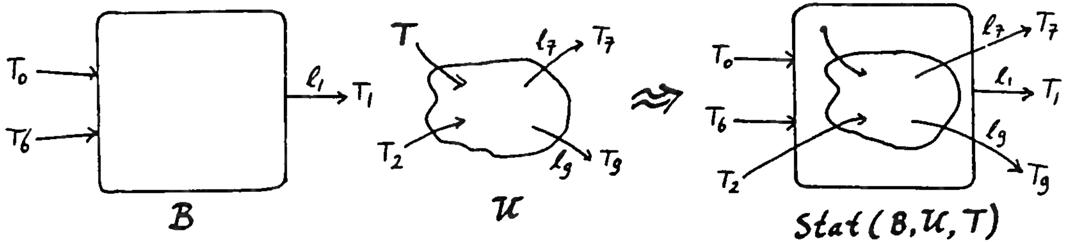


Figure 3: Statification

- Or-construct: $Or(U_1, U_2)$; leads to a statechart which becomes an OR-state after statification.

- And-construct: $And(U_1, U_2)$; yields an AND-state after statification.

In the constructs above both constituents should not have joint incoming or joint outgoing transitions with the same name, except for the AND-construct where joint incoming transitions are allowed.

- Connect: $Connect(U, T_1, T_2)$; results in a chart identical to U except that outgoing arc T_1 and incoming arc T_2 of U are connected to form a single complete transition.

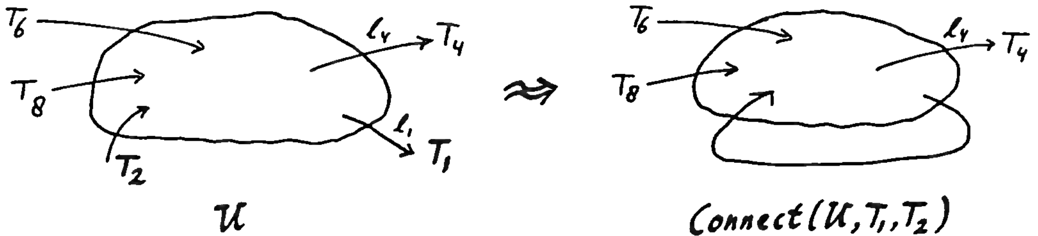


Figure 4: Connection

- Hide-Closure: $HiCl(U,a)$; hides any generation of a by U (Hiding) and makes U insensitive to any a generated by the environment (Closure).

After the informal introduction into the syntax of Statecharts above, we give the formal syntax. First we define the labels that can be associated with the transitions of any statechart and the event expressions used in these labels.

2.2.1 Events

Let E_e be a set of elementary/atomic events. The set of composite events Exp is defined inductively as the least set satisfying:

- $\lambda \in Exp$, $\neg\lambda \in Exp$.
- if $e \in E_p$ then $e \in Exp$, $\neg e \in Exp$.
- if $e \in Exp$, $n \in N \setminus \{0\}$ then $tm(e,n) \in Exp$, $\neg tm(e,n) \in Exp$.
- if $e_1, e_2 \in Exp$ then $e_1 \vee e_2 \in Exp$, $e_1 \wedge e_2 \in Exp$.

2.2.2 Transition Labels

The set of all symbols that can label the transitions of a statechart is the set **Lab** defined as follows:

$$\mathbf{Lab} = \{E/A \mid E \in Exp, A \subseteq E_e, A \text{ is finite}\}$$

If A is a singleton set then we often use the event itself, i.e. E/a abbreviates $E/\{a\}$.[4]

2.2.3 Formal Syntax of Statecharts

Let Σ be the set of all states (or more precisely state names) and $\mathcal{T}_I$ and $\mathcal{T}_O$ be the set of all (names of) incoming and outgoing transitions of any statechart such that $\mathcal{T}_I \cap \mathcal{T}_O = \emptyset$. Also let $L : \mathcal{T}_O \rightarrow \mathbf{Lab}$ denote the labeling function that labels all the outgoing transitions. Assume $\mathcal{T} = \mathcal{T}_I \cup \mathcal{T}_O$ and E_e are countable.

The set of statecharts is defined by the following BNF-grammar, where $a \in E_e$, $I \subseteq \mathcal{T}_I$, $O \subseteq \mathcal{T}_O$, I and O are finite, $\{T, T_2\} \subseteq \mathcal{T}_I$, $T_1 \in \mathcal{T}_O$, $S \in \Sigma$.

$$\begin{aligned}
U &::= Disj \mid Conj \\
Disj &::= Prim \mid \mathbf{Or}(Disj,Disj) \mid \mathbf{Connect}(Disj, \mathbf{T_1}, \mathbf{T_2}) \\
Conj &::= \mathbf{And}(Default,Default) \mid \mathbf{And}(Default,Conj) \\
Prim &::= Basic \mid Default \mid \mathbf{HiCl}(U,\mathbf{a}) \\
Default &::= \mathbf{Stat}(Basic,U,\mathbf{T}) \\
Basic &::= [\mathbf{I},\mathbf{O},\mathbf{S}]
\end{aligned}$$

[4] In the original syntax of labels as given in [HPPSS87], the action A is of the form $a_1, \cdots, a_n$ whereas we take A to be the set containing these events.

2.2.4 Syntactic Restrictions

There are certain syntactic conditions to be satisfied by any statechart. In order to describe these conditions we define two functions IN and OUT; for a given statechart U, IN(U) and OUT(U) are the sets of incoming and outgoing transitions of U, respectively.

	IN	OUT
$[I,O,S]$	I	O
$Stat(B,U,T)$	$IN(B) \cup IN(U) \setminus \{T\}$	$OUT(B) \cup OUT(U)$
$Connect(U,T_1,T_2)$	$IN(U) \setminus \{T_2\}$	$OUT(U) \setminus \{T_1\}$
$Or(U_1,U_2)$	$IN(U_1) \cup IN(U_2)$	$OUT(U_1) \cup OUT(U_2)$
$And(U_1,U_2)$	$IN(U_1) \cup IN(U_2)$	$OUT(U_1) \cup OUT(U_2)$
$HiCl(U,a)$	$IN(U)$	$OUT(U)$

Then we have the following syntactic restrictions:

- For $Connect(U,T_1,T_2)$: $T_1 \in OUT(U)$ and $T_2 \in IN(U)$.
- For $Stat(B,U,T)$: $T \in IN(U)$, $IN(B) \cap IN(U) = \emptyset$, and $OUT(B) \cap OUT(U) = \emptyset$.
- For $Or(U_1,U_2)$: $IN(U_1) \cap IN(U_2) = \emptyset$ and $OUT(U_1) \cap OUT(U_2) = \emptyset$.
- For $And(U_1,U_2)$: $OUT(U_1) \cap OUT(U_2) = \emptyset$.

Remarks:

1. In $And(U_1,U_2)$, the intersection of $IN(U_1)$ and $IN(U_2)$ need not be empty. Incoming arcs with identical names are 'merged'.
2. In $Stat(B,U,T)$, there should always be at least one incoming arc to U, to be taken as the default.
3. The *Concat* operation given in [HGR88] has not been provided in our syntax. It can be considered as a derived operation:
   $Concat(U_1,U_2,T_1,T_2) \equiv Connect(Or(U_1,U_2),T_1,T_2)$.

3 Denotational Semantics

As mentioned in the introduction, the semantic model associates with a statechart the set of all (maximal) computation histories representing complete computations. It has been shown in [HGR88] that, besides denotations for events generated at each computation step (the observables) and denotations for entry and exit, the following two additional denotations are necessary and sufficient to obtain a compositional semantics: (1) a set of all events assumed to be generated by the whole system (i.e. statechart together with its environment) at each step and (2) a causality relation between generated events. More precisely, a computation history h of a statechart U is of the form $h = (\hat{s}, i, f, o, s)$ where

- $\hat{s} \in N$ models the start step (N denotes the set of natural numbers).

- $i \in \mathcal{T}_I \cup \{\star\}$ is an incoming transition or $\star$ to model an implicit entry.
- $f : N \rightarrow \{(F, C, <) \mid F \subseteq C \subseteq E_e \text{ and } < \text{ a total order on } C\}$ records for every step n a triple $(F, C, <)$, where
  - F is a subset of the events generated by U. Considering the chain of transitions in step n, F contains the events which are generated by U, for the first time in this chain.
  - C is the set of events generated by the total system (i.e. U and its environment) in step n.
  - $<$ denotes the causal relationship between events generated by the whole system . If a causes b then $a < b$. If there is no causal relation, then the semantics of U will contain two histories: one with $a < b$ and another with $b < a$.
- $o \in \mathcal{T}_O \cup \{\star, \perp\}$ is an outgoing transition or $\star$ for an implicit exit, or $\perp$ when there is no exit.
- $s \in N \cup \{\infty\}$ denotes the exit step.

Henceforth, h will denote $(\hat{s}, i, f, o, s)$ and similarly for super- and sub-scripts:
h' denotes $(\hat{s}', i', f', o', s')$, h_1 denotes $(\hat{s}_1, i_1, f_1, o_1, s_1)$, etc.
For a function f as above, the fields of $f(n)$ are selected by $f^F(n)$, $f^C(n)$ and $f^<(n)$.
Define $\mathcal{H} = \{h \mid \hat{s} < s,\ o = \perp \leftrightarrow s = \infty, \text{ and } (v \leq \hat{s} \vee v > s) \rightarrow f^F(v) = \emptyset\}$.

Figure 5 shows why F is not equal to the set of *all* events generated by U. If a occurs

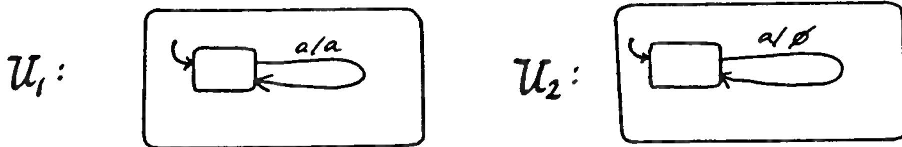


Figure 5:

externally, then U_1 generates a at every step, whereas U_2 does not generate a. This difference can not be sensed, however, by other statecharts, because a is generated in the system already. In order to get the same semantics for U_1 and U_2, both have an empty F-set to denote that both are not responsible for the first generation of a.

Our semantic domain is given by $(\mathcal{D}, \sqsubseteq)$, where $\mathcal{D} = \{H \mid H \subseteq \mathcal{H}\}$ and $D_1 \sqsubseteq D_2$ iff $D_1 \subseteq D_2$ for all $D_1, D_2 \in \mathcal{D}$. It is easy to show that our domain is a complete lattice with bottom element $\emptyset$ and top element $\mathcal{H}$. We give the semantics of statecharts by defining a semantic function $\mathcal{M}$ that maps any statechart to an element of $\mathcal{D}$, so to a set of histories. The semantics is an *a priori* semantics that anticipates an arbitrary environment.

Definition 3.1 (Basic) Any general computation history of a basic statechart $[I,O,S]$ enters the chart implicitly (denoted by $\star$) or via one of the arcs in I at a particular time step and starts waiting to exit the statechart from the next step onwards. Then there are three situations possible: it waits forever or it exits the chart at a finite step implicitly (also denoted by $\star$) or by taking one of the outgoing arcs in O; in the latter case, the necessary condition for taking the transition should be true.
Using the predicates *wait* and *fire*, defined below, the semantics is given as follows.

$$\mathcal{M}([I,O,S]) = \{h \in \mathcal{H} \mid i \in (I \cup \{\star\}) \wedge \forall v, \hat{s} < v < s : wait(O,v) \wedge$$
$$[o = \perp \vee (o = \star \wedge f^F(s) = \emptyset) \vee fire(O,s)]\}$$

Let the label of an outgoing transition $t \in O$ be given by: $L(t) = E_t/A_t$.
$wait(O,v)$ characterizes the situation in which none of the transition in O can be taken. Then none of the triggers of these transitions evaluate to true and no event is generated by the statechart. Consequently, *wait* is defined as follows:

$$wait(O,v) \equiv f^F(v) = \emptyset \wedge \bigwedge_{t \in O} \neg inC(E_t, v)$$

where $inC(E_t, v)$ expresses that E_t evaluates to true at step v. It is defined inductively as follows:

$$\begin{aligned}
inC(\lambda, v) &\equiv true \\
inC(e, v) &\equiv e \in f^C(v), \quad \text{for } e \in E_e \\
inC(\neg e, v) &\equiv \neg inC(e, v) \\
inC(e_1 \vee e_2, v) &\equiv inC(e_1, v) \vee inC(e_2, v) \\
inC(e_1 \wedge e_2, v) &\equiv inC(e_1, v) \wedge inC(e_2, v) \\
inC(tm(e,n), v) &\equiv \begin{cases} inC(e, v-n) \wedge \forall v', v-n < v' < v : \neg inC(e, v') & \text{if } v \geq n \\ false & \text{otherwise} \end{cases}
\end{aligned}$$

Predicate $fire(O,v)$ describes the condition for taking a transition $t \in O$ at step v. Then its trigger E_t evaluates to true, so $inC(E_t, v)$ must hold and all the events in A_t are generated. Furthermore, certain causal relations exist between newly generated events and the events that triggered the transition. These relations are expressed by predicate $rel(E_t, a, v)$, defined below. Consequently, *fire* is defined as follows:

$$fire(O,v) \equiv \bigvee_{t \in O} (o = t \wedge inC(E_t, v) \wedge f^F(v) \subseteq A_t \subseteq f^C(v) \wedge$$
$$\bigwedge_{a \in A_t} a \in f^F(v) \rightarrow rel(E_t, a, v))$$

Predicate $rel(E_t, a, v)$ provides the necessary causal relation between an event $a \in A_t$ and the events in E_t. For instance, if $E_t \equiv b$ then we need $(b,a) \in f^<(v)$, whereas for $E_t \equiv \neg b$

there should not be any relation between a and b because b does not occur in step v. We define rel inductively as follows:

$$\begin{aligned}
rel(\lambda, a, v) &\equiv true \\
rel(b, a, v) &\equiv \begin{cases} (b,a) \in f^{<}(v), & \text{if } b \not\equiv a \\ false & \text{if } b \equiv a \end{cases} \\
rel(tm(e,n), a, v) &\equiv inC(tm(e,n), v) \\
rel(\neg e, a, v) &\equiv \neg inC(e, v) \\
rel(e_1 \vee e_2, a, v) &\equiv rel(e_1, a, v) \vee rel(e_2, a, v) \\
rel(e_1 \wedge e_2, a, v) &\equiv rel(e_1, a, v) \wedge rel(e_2, a, v)
\end{aligned}$$

Definition 3.2 (Or) The semantics of a *Or* construct is the union of the semantics of its constituents.

$$\mathcal{M}(Or(U_1, U_2)) = \mathcal{M}(U_1) \cup \mathcal{M}(U_2)$$

Definition 3.3 (Connect) Execution of $Connect(U, T_1, T_2)$ consists of first (a) entering U via an arc other than T_2, and then (b) taking transitions as specified by U, indefinitely, or exiting U either via an arc other than T_1, or exiting via T_1, re-entering U via T_2, and repeating (b). Given two sets of histories D_1, D_2, we define $CONC(D_1, D_2, T_1, T_2)$ (informally) as the set of (i) histories from D_1 which do not exit via T_1, and (ii) histories which consist of a history from D_1 exiting via T_1 followed by a history from D_2 entering via T_2.[5]

$$\begin{aligned}
&CONC(D_1, D_2, T_1, T_2) = \\
&\quad \{h \mid h \in D_1 \wedge o \neq T_1\} \cup \\
&\quad \{h \mid \exists h_1 \in D_1, h_2 \in D_2 : \hat{s} = \hat{s}_1 \wedge s_1 = \hat{s}_2 \wedge s = s_2 \wedge i = i_1 \wedge i_2 = T_2 \wedge \\
&\qquad o_1 = T_1 \wedge o = o_2 \wedge f^F = f_1^F \cup f_2^F \wedge f^C = f_1^C = f_2^C \wedge f^< = f_1^< = f_2^<\}
\end{aligned}$$

Then $\mathcal{M}(Connect(U, T_1, T_2))$ can be obtained by removing the histories with a T_2 entry from the largest set satisfying $D = CONC(\mathcal{M}(U), D, T_1, T_2)$, i.e., the greatest fixed point $\nu_X.CONC(\mathcal{M}(U), X, T_1, T_2))$. (Note that such a set D will not contain histories which exit via T_1, because a T_1-exit leads—by (b) above—to a T_2-entry into U.) It is easy to see that $CONC$ is monotonic in its second argument and hence has a greatest fixed point in our complete lattice (see e.g. [dB80]). This leads to

$$\mathcal{M}(Connect(U, T_1, T_2)) = del_{T_2}(\nu_X.CONC(\mathcal{M}(U), X, T_1, T_2))$$

where $del_{T_2}(D) = \{h \in D \mid h = (\hat{s}, i, f, o, s) \wedge i \neq T_2\}$.

The semantics can also be given as the intersection of approximations:

$$\mathcal{M}(Connect(U, T_1, T_2)) = del_{T_2}(\bigcap_{k \in N} D_k)$$

with $D_0 = \mathcal{H}$, and for $k \geq 0$: $D_{k+1} = CONC(\mathcal{M}(U), D_k, T_1, T_2))$.

[5] $CONC$ stands for concatenation.

So the semantics consists of all those histories that exit and re-enter U through the connected arc for a finite/infinite number of times. All the finite histories eventually exit U via an arc other than T_1.

Definition 3.4 (And) A history h from the semantics of $And(U_1, U_2)$ is obtained by combining h_1 from $\mathcal{M}(U_1)$ and h_2 from $\mathcal{M}(U_2)$, provided certain conditions are fulfilled. Since all orthogonal components of an And-construct are entered and exited simultaneously, the entry steps, $\hat{s}_1$ and $\hat{s}_2$, must be equal, and also the exit steps, s_1 and s_2. Furthermore, the claims in h_1 and h_2 about the total system—represented by the C and $<$ components—must be the same. The incoming transition in h, component i, can be either

- a $\star$, denoting an implicit entry, if both i_1 and i_2 are $\star$, or
- a joint incoming transition, so $i = i_1 = i_2$, or
- the incoming transition of one, provided the other is $\star$.

The o component in h, describing the way in which $And(U_1, U_2)$ is exited, can be either

- a $\star$, if $o_1 = o_2 = \star$, denoting an implicit exit, or
- an outgoing transition of one, provided the other is $\star$, or
- a $\perp$, to denote that $And(U_1, U_2)$ is never exited, if both o_1 and o_2 are $\perp$.

This leads to the following definition:

$$\begin{aligned}\mathcal{M}(And(U_1,U_2)) = \{h \mid \exists h_1 \in \mathcal{M}(U_1), h_2 \in \mathcal{M}(U_2) : \hat{s} = \hat{s}_1 = \hat{s}_2 \wedge s = s_1 = s_2 \wedge \\ f^C = f_1^C = f_2^C \wedge f^< = f_1^< = f_2^< \wedge f^F = f_1^F \uplus f_2^F \wedge \\ [(i = i_1 = i_2) \vee (i = i_1 \wedge i_2 = \star) \vee (i = i_2 \wedge i_1 = \star)] \wedge \\ [(o = o_1 \neq \perp \wedge o_2 = \star) \vee (o = o_2 \neq \perp \wedge o_1 = \star) \vee o = o_1 = o_2 = \perp]\}\end{aligned}$$

Definition 3.5 (Statification) The semantics of $Stat(B, U, T)$ is similar to $\mathcal{M}(And(B, U))$, except for the way in which the statified chart is entered; any entry to B leads to U via the default arc T and the direct entry to T is no longer possible. Consequently, the semantics is given as follows:

$$\begin{aligned}\mathcal{M}(Stat(B,U,T)) = \{h \mid \exists h_1 \in \mathcal{M}(B), h_2 \in \mathcal{M}(U) : \hat{s} = \hat{s}_1 = \hat{s}_2 \wedge s = s_1 = s_2 \wedge \\ f^C = f_1^C = f_2^C \wedge f^< = f_1^< = f_2^< \wedge f^F = f_1^F \uplus f_2^F \wedge \\ [(i = i_1 \wedge i_2 = T) \vee (i = i_2 \neq T \wedge i \neq \star \wedge i_1 = \star)] \wedge \\ [(o = o_1 \neq \perp \wedge o_2 = \star) \vee (o = o_2 \neq \perp \wedge o_1 = \star) \vee o = o_1 = o_2 = \perp]\}\end{aligned}$$

Definition 3.6 (Hide and Close) In the semantics of $HiCl(U, a)$ we first require that every occurrence of a is generated by U. Thereafter a is hidden by allowing arbitrary behaviour for it in the new histories as far as the C and $<$ component are concerned, and removing a from the F component. First we define the *pointwise subtraction* of a set-valued function g and a set A, notation $g \dot{-} A$, as follows, $(g \dot{-} A)(v) = g(v) - A$, for all $v \in N$. For a function $f^<$ s.t. $f^<(v) \subseteq E_e \times E_e$, let $f^<|_a$ be defined as $f^<|_a = f^< \dot{-} (E_e \times \{a\}) \dot{-} (\{a\} \times E_e)$. Then the semantics is given by

$$\mathcal{M}(HiCl(U,a)) = del_a(\{h \mid h \in \mathcal{M}(U) \wedge \forall v : a \in f^C(v) \rightarrow a \in f^F(v)\})$$

where $del_a(D) = \{h \in \mathcal{H} \mid \exists h_1 \in D : \hat{s} = \hat{s}_1 \wedge i = i_1 \wedge o = o_1 \wedge s = s_1 \wedge$
$$f^F = f_1^F \dot{-} \{a\} \wedge f^<|_a = f_1^<|_a \wedge f^C \dot{-} \{a\} = f_1^C \dot{-} \{a\}\}.$$

4 Related Work

A denotational semantics has been given for the graphical, state-based, specification language Statecharts. This semantics serves as a basis for a compositional axiomatisation of Statecharts in terms of a logic which is strong enough to express both safety and liveness properties. In contrast with [HGR88], we did not aim at full abstractness; our purpose was to give the semantics of a small subset of Statecharts while maintaining the essentials of a event-driven synchronous language. The compositional syntax and the main primitives of our denotations are derived from [HGR88]. An operational, non-compositional, semantics for Statecharts has been given in [HPPSS87].

In [Gon88] a related denotational semantics has been given for the non-graphical synchronous language Esterel. An operational description for this language can be found in [BC85]. Related real-time models have been given for extensions of CSP. [KSR+88] contains a denotational semantics for a real-time version of CSP, based on the linear history semantics of [FLP84]. Huizing extends the same model to achieve a fully abstract semantics [HGR87] for an OCCAM-like language. Reed and Roscoe [RR88] give a hierarchy of timed models for CSP, based on a complete metric space structure. A fully abstract timed failure semantics for an extended CSP language has been developed in [GB87].

References

[BC85] B. Berry and L. Cosserat. The synchronous programming language Esterel and its mathematical semantics. In *Proceedings CMU Seminar on Concurrency*, pages 389–449. LNCS 197, Springer-Verlag, 1985.

[BCH85] J.-L. Bergerand, P. Caspi, and N. Halbwachs. Outline of a real-time data flow language. In *Proceedings IEEE Real-Time Systems Symposium*, 1985.

[dB80] J. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[FLP84] N. Francez, D. Lehman, and A. Pnueli. A linear history semantics for distributed programming. *Theoretical Computer Science*, 32, 1984.

[GB86] P. le Guernic and A. Benveniste. Real-time, synchronous, data-flow programming: The language signal and its mathematical semantics. Technical Report 620, INRIA, Rennes, 1986.

[GB87] R. Gerth and A. Boucher. A timed failures model for extending communicating processes. In *Proceedings in the 14th International Colloquium on Automata, Languages and Programming*, pages 95–114. LNCS 267, Springer-Verlag, 1987.

[Gon88] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones; Application à ESTEREL*. PhD thesis, University of Orsay, 1988.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31:514 – 530, 1988.

[HGR87] C. Huizing, R. Gerth, and W.P. de Roever. Full abstraction of a real-time denotational semantics for an OCCAM-like language. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 223–237, 1987.

[HGR88] C. Huizing, R. Gerth, and W.P. de Roever. Modelling statecharts behaviour in a fully abstract way. In *Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, pages 271–294. LNCS 299, Springer-Verlag, 1988.

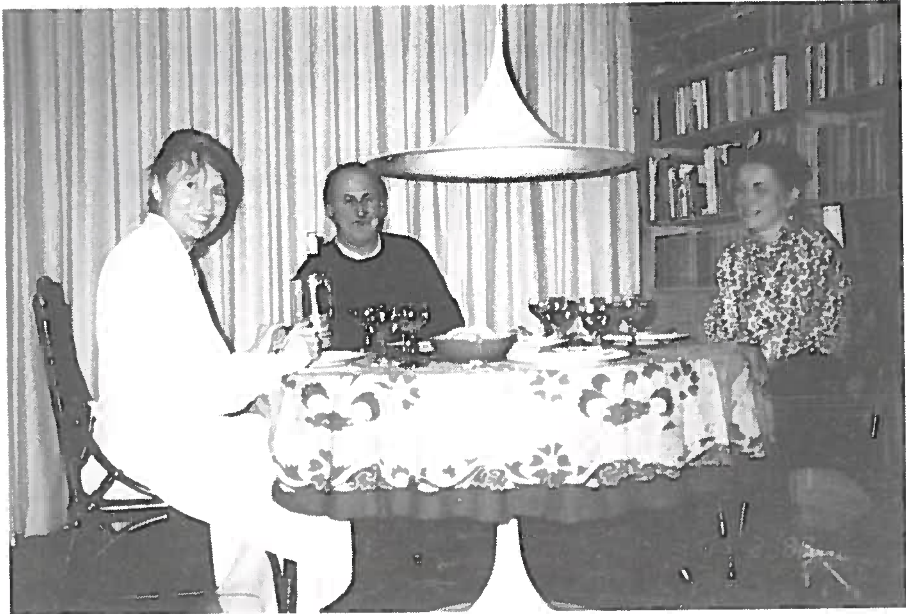
[HPPSS87] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings Symposium on Logic in Computer Science*, pages 54–64, 1987.

[JM89] F. Jahanian and A. Mok. Modechart, a specification language for real-time systems. *IEEE Transactions on Software Engineering*, to appear, 1989.

[KSR+88] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79(3):210–256, 1988.

[PS88] A. Pnueli and M. Shalev. What is in a step. Draft, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 1988.

[RR88] G. Reed and A. Roscoe. Metric spaces as models for real-time concurrency. In *Proceedings of the 3th Workshop on the Mathematical Foundations of Programming Languages Semantics 87*, 1988.



Beste Jaco en Angeline,

Allereerst onze hartelijke gelukwensen met het 25-jarig jubileum van Jaco's dienstverband bij het Mathematisch Centrum. Wij hopen dat Jaco nog vele jaren de steunpilaar zal blijven die hij nu al zolang voor het CWI en de Informatica-gemeenschap is.

Wat onze bijdrage aan de Semantiek betreft, wij zijn niet verder gekomen dan de formule AA BB A:

"Daar wij niet zo sterk zijn in semantiek,
op deze wijze onze repliek.
Het gaat over wijn,
gezellig samen zijn.
Geen infor- maar romantiek."

Aly en Piet van der Houwen.

Scanner generation for modular regular grammars

P. Klint
Department of Software Technology, Centre for Mathematics and Computer Science
Programming Research Group, University of Amsterdam

When formal language definitions become large it may be advantageous to divide them into separate modules. Such modules can then be combined in various ways, but this requires that implementations derived from individual modules can be combined as well. In this paper we address the problem of combining regular grammars appearing in separate modules and of combining the lexical scanners generated for them.

Dedicated to J.W. de Bakker on the occasion of the 25th anniversary of his association with the Centre for Mathematics and Computer Science.

Key Words & Phrases: program generator, generation of lexical scanners, modular regular grammars, finite automata, subset construction.

1987 CR Categories: D.1.2 [**Programming Techniques**]: Automatic programming; D.3.4 [**Programming Languages**]: Processors.

1985 Mathematics Subject Classification: 68N20 [**Software**]: Compilers and generators.

1. INTRODUCTION

The benefits of dividing complex systems into several, smaller, modules are well-known. Apart from a reduction in complexity that can be achieved for individual modules, one also introduces the possibility of re-using a module several times. In this paper we will apply the idea of modular decomposition to the definition of formal languages (such as, e.g., programming languages and specification languages), and concentrate on *lexical syntax*, one of the syntactic aspects that have to be defined for a language. Typically, the lexical syntax defines comment conventions, layout symbols, and the form of identifiers, keywords, delimiters, and constants (e.g. numbers, strings) in a language. The standard method is now to use a regular grammar to specify the precise form of the various elements in the lexical syntax and to compile this regular grammar into a deterministic finite state automaton (DFA) to be used for the actual reading of program texts.

Here, we are interested in the problem of how the lexical syntax can be subdivided in separate modules and how DFAs can be obtained from various combinations of these modules. The motivation for this problem comes from two different sources:

- In a setting where definitions for more than one language are being developed, it is natural to construct a set of standard modules defining frequently used notions (e.g., identifiers, floating point numbers, or string constants). Of course, these standard notions may sometimes need adaptation depending on their use (e.g., the letters

```
Abbreviations:

    M1:  <DIGIT>   = 0 | 1 | ... | 7
    M2:  <DIGIT>   = 8 | 9
    M3:  <LETTER>  = a | b | ... | z

Rules:

    M4:  <INT>     = <DIGIT>+
    M5:  <REAL>    = <INT> "." <INT>
    M6:  <ID>      = <LETTER> (<LETTER> | <DIGIT>)*
    M7:  <KW>      = if
    M8:  <KW>      = end
```

Figure 1. A modular regular grammar.

appearing in identifiers may or may not contain both lower case letters and upper case letters, integer constants may or may not contain hexa-decimal digits).

- In modular specification languages that allow user-definable syntax to be introduced in each module, the composition of modules requires, among others, the composition of lexical syntax.

The flexibility we want to achieve can best be illustrated by an example. Consider the grammar shown in Figure 1. It consists of two parts: abbreviations and rules. The abbreviations part defines named regular expression to be used in the rules part. When several regular expressions e_i are associated with one name, we associate with that name a regular expression containing all expressions e_i as alternatives. The actual regular grammar is defined in the rules part. Names appearing in rules can be completely eliminated by textual substitution. The names of rules define the token-name to be associated with a string recognized by that particular rule. Note that more than one rule may recognize the same string; in that case we associate more than one token-name with it.

In the example, we define a lexical syntax containing integer constants, real constants, identifiers and the keywords `if` and `end`. Each regular expression in the grammar is labelled with a module name. In general, several expressions may be labelled with the same name, but in this example we have the extreme case that every expression is labelled with a different name. The use of this modular regular grammar is shown in Figure 2. Given a list of selected module names, only those rules are to be used whose module name appears in the selection. For each selection of modules, the modular regular grammar thus corresponds to a (probably) different ordinary regular grammar.

An implementation of modular regular grammars should, clearly, have the following two properties:

- The time needed to construct a DFA for a given selection of modules (in the modular case) should be significantly less than the time needed to construct the automaton from scratch (in the non-modular case) using only the rules from the selected modules.
- The efficiency of the DFA generated in the modular and in the non-modular case should be comparable.

modules	selection (1)	selection (2)	selection (3)	selection (4)
M1	x	x	x	x
M2	x	o	x	x
M3	x	x	x	x
M4	x	x	o	x
M5	x	x	x	x
M6	x	x	x	o
M7	x	x	x	x
M8	x	o	x	x
sentences	**recognized as**	**recognized as**	**recognized as**	**recognized as**
123	<INT>	<INT>	-	<INT>
678	<INT>	-	-	<INT>
2.8	<REAL>	-	-	<REAL>
abc	<ID>	<ID>	<ID>	-
end	<ID>,<KW>	<ID>	<ID>,<KW>	<KW>
xy9	<ID>	-	<ID>	-

Figure 2. Examples of module selections.

How can modular regular grammars now be compiled into DFAs? There are two, fundamentally different, solutions to this problem:

- Compile all rules that are labelled with the same module name into a single DFA and define a composition operation on DFAs. The DFA constructed for a certain selection of modules then consists of the composition of the DFAs constructed for each individual module in the selection.
- Compile all rules into a single DFA and define a selection operation that, given a list of selected modules, extracts the sub-automaton that corresponds to that selection.

Obviously, the first solution is the most elegant one since it leads to a truly modular implementation of lexical scanners. Unfortunately, the composition operation on deterministic automata is expensive: given two DFAs A and B and their composition $A \cup B$, in many cases the states of A and B do not appear in $A \cup B$. Instead, they are combined into new states thus reflecting the interactions between the languages recognized by A and B. As a result, the computation of $A \cup B$ requires roughly the same amount of work as the construction of a completely new automaton. This is illustrated in Figure 3, where the DFAs for the two regular expressions `a b`, and `(a | c) d` are shown together with the resulting DFA for the combined language { `a b`, `(a | c) d` } or, equivalently, `(a b) | ((a | c) d)`. It should be emphasized that the complexity of the composition operation is caused by our (efficiency) requirement that the result of the composition is again a *deterministic* automaton. Allowing a non-deterministic automaton (NFA) as result, would significantly simplify the composition operation as is shown in Figure 4.

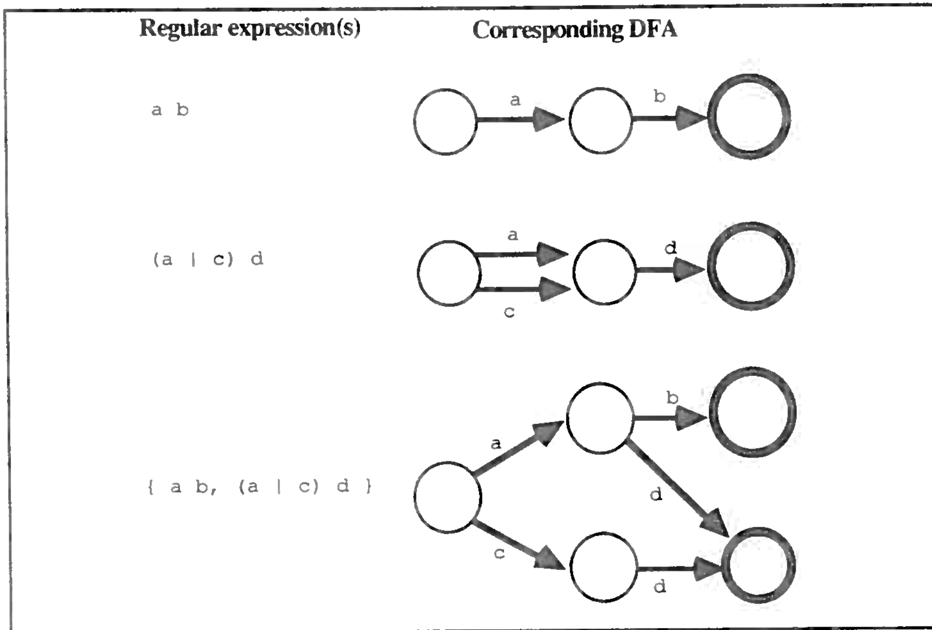


Figure 3. Two DFAs and their composition as DFA.

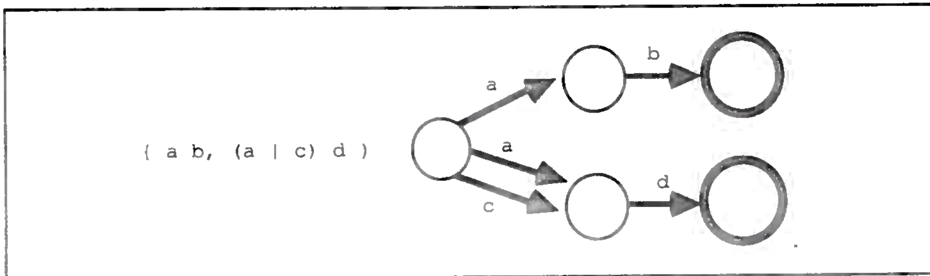


Figure 4. Same DFAs and their composition as NFA.

In this paper, we now concentrate on the second solution mentioned above and investigate how a selection operation can be defined on a DFA that extracts a sub-automaton corresponding to a selection of modules.

2. AN ALGORITHM FOR COMPILING REGULAR EXPRESSIONS

In this section we sketch an algorithm for the lazy compilation of regular expressions into deterministic finite automata. A complete description of this method can be found in [HKR87b].

2.1. Preliminaries

First, we introduce the notions of *regular expression* and *labelled regular expression.*

Regular expressions over a finite alphabet Σ are composed of the symbols from that alphabet, the empty string (ε), the operators concatenation (denoted by juxtaposition), alternation (|), repetition (*), and parentheses. We will adopt the convention that parentheses may be omitted under the assumption that the operators in regular expressions are left associative and that * has the highest priority, concatenation has the second highest priority and | has the lowest priority. We will also use the undefined regular expression ($\perp$) denoting the empty set of strings, i.e. $\perp$ does not recognize anything. The following identities characterize the interactions between concatenation, |, * and $\perp$:

(a) $r \perp = \perp r = \perp$

(b) $\perp^* = \varepsilon$

(c) $r \mid \perp = \perp \mid r = r$

A *labelled regular expression* is a regular expression in which a unique natural number p is associated with each occurrence of a symbol $a \in \Sigma$. We say that a occurs at position p and that the symbol at position p is a, notation: a_p. Also define $symbol(p)=a$ for each a_p.

We will use some auxiliary functions on labelled regular expressions which describe properties of the strings recognized by them:

(1) The predicate *nullable* determines whether a regular expression can recognize the empty string.

(2) The function *firstpos* maps a labelled regular expression to the set of positions that can match the first symbol of an input string.

(3) The function *lastpos* maps a labelled regular expression to the set of positions that can match the last symbol of an input string.

(4) The function *followpos* maps a position in a labelled, regular expressions e to the set of positions that can follow it, i.e., if p is a position with $symbol(p) = a$ and p matches the symbol a in some legal input string $...ab...$, then b will be matched by some position in $followpos(p, e)$.

For precise definitions of these functions we refer the reader to [HKR87b] or [BS87].

In the sequel, we will adopt the convention that a unique symbol $\$ \in \Sigma$ is used to terminate both regular expressions and input strings. A *terminated, labelled*, regular expression e over an alphabet Σ, has the form $e'\$$, where e' is a labelled regular expression over $\Sigma \backslash \{\$\}$.

An *accepting sequence of positions* for a labelled regular expression e can now be defined as a sequence of positions $p_1,...,p_n$ such that $p_1 \in firstpos(e)$, $p_n \in lastpos(e)$, and $p_{i+1} \in followpos(p_i, e)$, $i=1,...,n-1$. For all strings $s \in \Sigma^*$ and for all terminated, labelled, regular expressions e over Σ the following holds: $s=a_1...a_n$ with $a_n = \$$ belongs to the set of strings denoted by e if and only if there exists an accepting sequence of positions $p_1,...,p_n$ for e such that $a_i = symbol(p_i)$, $i=1,...,n$. (see [YM60], Theorem 3.1).

2.2. Algorithms for the lazy construction of a DFA

Using the notions introduced in the previous section we now formulate an algorithm for the lazy construction of a deterministic finite automaton for a given set of regular expressions. The basic idea is to construct a deterministic automaton in which each state corresponds to a *set* of positions in the set of regular expressions. In this way, each state may represent *several* ways of recognizing an input string. The initial state of the automaton consists of the first positions of all the regular expressions. Transitions from the start state, as well as from any other state, are computed as follows: consider for each symbol a in the alphabet (or the end marker) the positions that can be reached when recognizing a in the input; the

set of positions that can be reached in this way form the (perhaps already existing) state to which a transition should be made from the original state on input a. The set of positions that corresponds to a state thus characterizes the progress of all possible accepting sequences for input strings with a common head.

In principle, the powerset of all positions in the set of regular expressions should be considered during the construction of a DFA. The following algorithms only consider the sets of positions that are really used during this construction. These sets are collected in the set *States*. When a state S is added to *States*, it is unexpanded and *expanded*(S) = **false** holds. A state $S \in States$ can be marked as expanded by setting *expanded*(S) := **true**.

The DFA that is being constructed is represented by an initial state $start \in States$ and a transition function $Trans : States \times \Sigma \rightarrow States$.

In standard DFA construction algorithms, a complete DFA is computed for a given regular expression. In the following lazy algorithm only a Partial DFA (PDFA) is constructed which is further extended when needed during scanning of given input strings.

First, we give the algorithms for the lazy construction of the start state and for the expansion of a state.

Algorithm *L-CONSTRUCT*

Construction of the initial part of the DFA that accepts the language described by a set of regular expressions.

Input. A set E of terminated, labelled, regular expressions over alphabet Σ.

Output. A PDFA in which only the start state has been expanded.

Method.

$A.start := \bigcup_{e \in E} firstpos(e)$

$A.States := \{ A.start \}$
$A.Trans := \varnothing$
return(*EXPAND*(E, A, $A.start$))

Algorithm *EXPAND*

Expansion of a PDFA state.

Input. A set of terminated, labelled, regular expressions E, a corresponding PDFA A, and a state S.

Output. The original PDFA expanded with all states to which S has transitions, and a definition of these transitions.

Method.

for $\forall\, a \in \Sigma \backslash \{\$\}$
do
$U := \bigcup_{\{p \in S \mid symbol(p) = a\}} followpos(p, E)$

if $U \neq \varnothing \wedge U \notin A.States$ **then** $A.States := A.States \cup \{U\}$ **fi**
$A.Trans(S,a) := U$
od
expanded(S) := **true**
return(A)

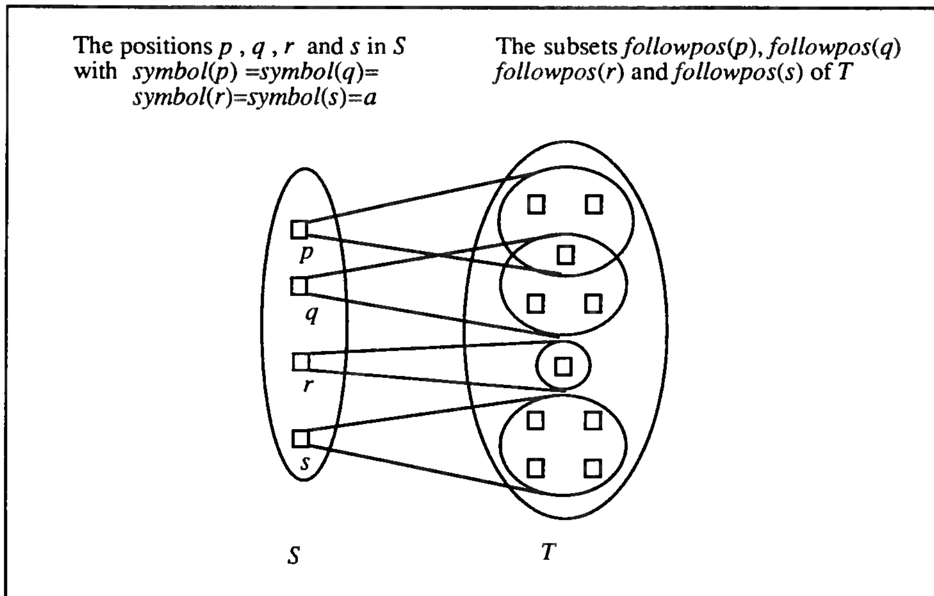


Figure 5. Positions causing a transition between states S and T on symbol a.

For later reference it is useful to emphasize that the existence of a transition between two states S and T on alphabet symbol a may be caused by *several* positions in S that correspond to the symbol a. State T will contain as subsets the, possibly overlapping, sets of follow positions for each of these positions in S. This situation is illustrated in Figure 5.

From the definition of *EXPAND* it follows that a state can never correspond to an empty set of positions. For convenience, we will assume in the sequel that all automata contain an *error state* with the following properties:

1. The error state corresponds to the empty set of positions.
2. The error state is not an accepting state.
3. The transition function is augmented as follows:

   (a) for each state, transitions to the error state are added for all characters in Σ for which that state has no legal transition.

   (b) for all characters in Σ, the transition function contains a transition from the error state to itself.

These additions to the generated automata are implicit and will not be shown in the diagrams.

Finally, we give the scanning algorithm associated with *L-CONSTRUCT*. It performs expansions of needed, unexpanded, states.

Algorithm *L-SCAN*

Simulate a given PDFA on a given input string, incrementally expanding the PDFA when necessary.

Input. A set E of terminated, labelled, regular expressions, a corresponding PDFA A, and an input sentence $s= a_1...a_n$, with a_n=$.
Output. **true** or **false** (indicating acceptance or rejection of the input string) and a possibly extended version of A.
Method.

```
S:= A.start
i := 1
while a_i ≠ $
do
   if ¬ expanded(S) then A := EXPAND(E, A, S) fi
   S:= A.Trans(S, a_i)
   i := i + 1
od
return (FINAL(S), A)
```

The last state reached during the scanning of an input string determines whether the input string should be accepted or rejected. A state is accepting if one of its positions corresponds to the end marker $. This is defined by the following algorithm.

Algorithm *FINAL*
Determine whether a given state is an accepting state.
Input. A state S.
Output. **true** or **false**
Method.

return $\exists\, p \in S\ [symbol(p) = \$]$

Note that a state may contain several positions with symbol $. This may happen when a string is recognized by more than one rule in the regular grammar.

3. AN ALGORITHM FOR COMPILING MODULAR REGULAR GRAMMARS

3.1. Modular regular expressions versus modular regular grammars

Before generalizing these lazy scanner generation techniques to the case of modular regular grammars, we first need a definition of modular regular grammars. It would be natural to describe them as (module-name, regular expression) pairs. However, it turns out that not all sub-expressions of a regular expression need to originate from the same module. This will become clear when discussing named regular expressions in Section 3.3. Therefore, we choose a method that allows more refined control over the module information and associate module names with the *positions* in a (terminated, labelled) regular expression and not with the regular expression as a whole. We will write *module*(p) to denote the module name associated with position p and we will write ${}_m a_p$ to denote a position p such that *symbol*(p)=a and *module*(p)=m. We will call these regular expressions with associated module information *modular regular expressions*. In this section, we will only use sets of modular regular expressions. In Section 3.3, *modular regular grammars* will be introduced and we will show how they can be reduced to sets of modular regular expressions.

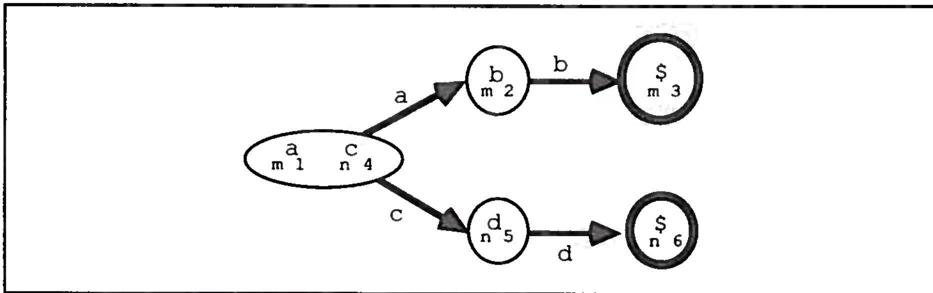


Figure 6.

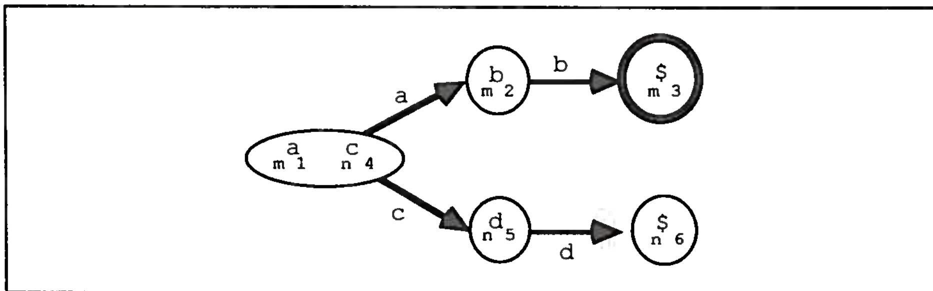


Figure 7.

Given a modular regular expression e and a list of module names M we can now *restrict* expression e to M (notation: e/M) by replacing all ${}_m a_p$ in e with $m \notin M$ by the undefined expression $\perp$. We extend the restriction operator / to sets of regular expressions.

The problem we want to solve can now be formulated as follows: given a set of modular regular expressions E, a partially constructed automaton A for these regular expressions, and a list of module names M, can we select a part of A that precisely recognizes the language defined by E restricted to M?

The simplest method one can imagine to restrict the language accepted by a given DFA is to use the DFA as it is, but impose restrictions on accepting states according to the current selection of modules. This method would only require some recomputations on the accepting states of the DFA.

Consider, for instance, the set of expressions $E = \{ {}_m a_1 \; {}_m b_2 \; {}_m \$_3 \,,\; {}_n c_4 \; {}_n d_5 \; {}_n \$_6 \}$ and the corresponding DFA A shown in Figure 6 (the border lines of accepting states are shown in bold face). Choosing the set of modules $\{m\}$, $E/\{m\}$ is then equal to $\{ {}_m a_1 \; {}_m b_2 \; {}_m \$_3 \,,\; \perp \}$ $= \{ {}_m a_1 \; {}_m b_2 \; {}_m \$_3 \}$ and the automaton obtained from A by only retaining accepting states that are labelled with a position in the selection $\{m\}$ correctly recognizes the language defined by $E/\{m\}$ (see Figure 7).

However, on closer inspection it turns out that this simple method may be incorrect when the positions in a single modular regular expression are labelled with different module names. This is illustrated by the following counter example. Consider

$E = \{ \, (\, {}_m a_1 \mid {}_n b_2 \,) \; {}_n c_3 \; {}_n \$_4 \, \}$

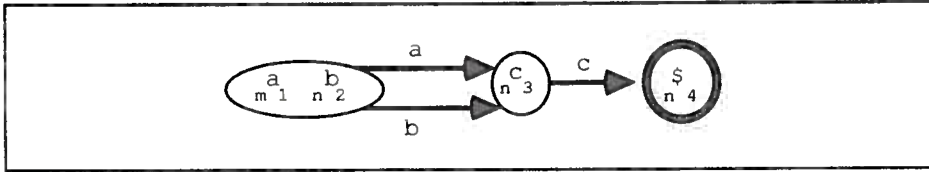


Figure 8.

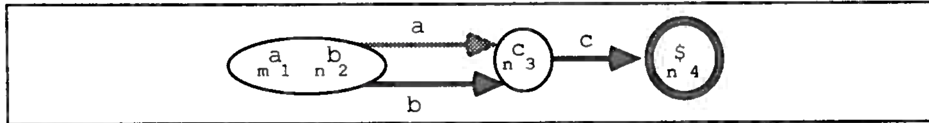


Figure 9.

with corresponding DFA shown in Figure 8. When we restrict E to the single module {n} we obtain

$$E/\{n\} = \{ (\perp \mid {}_n b_2) \; {}_n c_3 \; {}_n \$_4 \} = \{ {}_n b_2 \; {}_n c_3 \; {}_n \$_4 \}$$

but the string ac will (erroneously) be accepted using the simple method of restricting the accepting states of the DFA corresponding to E. The reason is, of course, that it is not sufficient to require that the accepting position is in the current selection of modules, as long as it can be reached using transitions that do *not* belong to that selection, e.g. the transition from the start state on symbol a. Therefore, we should remove all such invalid transitions obtaining the DFA shown in Figure 9 (invalid transitions are represented by shaded arrows).

Following this second method, we restrict the language accepted by a given DFA by eliminating all transitions in the DFA that do not correspond to the regular expressions in the current selection. This method requires the calculation of modifications to the transition table of the DFA for each new selection of modules.

3.2. Restricting a PDFA to a selection of modules

Given a PDFA A and a list of selected modules M, we have to compute those parts of the transition table that are still valid in this new selection. We introduce the following notions to achieve this goal:

(1) The table containing the transitions that are valid in the current selection will be called *SelTrans*, it is always a subset of the complete (but perhaps only partially computed) transition table *Trans* of A.

(2) With each state S in A we associate an attribute *specialized*, indicating whether the valid transitions leaving S have already been recorded in *SelTrans*. Initially, *specialized*(S) = **false** holds.

Remains the problem of formulating criteria to decide when a transition between two states S and T on alphabet symbol a is still valid in the current selection of modules. Our goal is to restrict the automaton A in such a way that it is equivalent to an automaton A' that would have been constructed when using the restricted set of regular expressions right from the start. In other words a transition should be valid in the restricted automaton A when it would have been constructed in A' as well. Looking at the way expansion of states is de-

fined (see Section 2.2, algorithm *EXPAND*) we observe that the existence of a transition between S and T on symbol a implies that

(1) S contains a position p whose symbol a;

(2) T contains some position q in the set of follow positions of p.

In the restricted automaton one should impose the additional restriction that positions p and q are labelled with module names appearing in the current selection.

Referring to Figure 5 given in Section 2.2, we illustrate the situation in the modular case in Figure 10. Positions that are labelled with a module in the current selection are indicated by a black square. A transition between S and T on symbol a is valid in this particular selection, since position q is selected and *followpos*(q) contains a selected position as well. In this case, q is the only position that supports this transition! Note that states S and T correspond to states S' and T' in automaton A' that contain *only* these selected positions.

These ideas are described more precisely in the following algorithm.

Algorithm *SPECIALIZE*

Input. A set E of terminated labelled regular expressions, a PDFA A, a state S, and a list of modules M.

Output. A modified version of A in which all valid transitions from state S in the modules in M have been recorded in *A.SelTrans*.

Method.

```
for ∀ a∈ Σ\{$}
do
   if ∃ p∈S , q∈A.Trans(S,a)
          [symbol(p) = a ∧ module(p)∈ M ∧ module(q)∈ M  ∧ q∈followpos(p,E)])
   then A.SelTrans(S,a) := A.Trans(S,a) fi
   specialized(S) := true
od
return(A)
```

Remains to be described how specialization and expansion of states interact during scanning. Before actual scanning starts, all states are set to unspecialized[1]. During scanning states are encountered that are either not yet expanded (and should be both expanded and specialized) or expanded but not yet specialized (and should be specialized). The algorithm is as follows:

Algorithm *M-SCAN*

Simulate a given PDFA for a given selection of modules on a given input string, incrementally expanding and specializing the PDFA when necessary.

Input. A set E of terminated, labelled, regular expressions, a corresponding PDFA A, a list of modules M, and an input sentence $s = a_1 \ldots a_n$, with a_n=\$.

Output. **true** or **false** (indicating acceptance or rejection of the input string) and a possibly extended version of A.

[1] Instead of setting all states to unspecialized at the beginning of M-SCAN, it would be more efficient to do this once in a separate selection operation. Subsequent applications of M-SCAN could then profit from the gradually increasing number of already specialized states.

Method.

for $\forall$ *State* $\in A.States$ **do** *specialized(State)* := **false od**;
$S := A.start$
$i := 1$
while $a_i \neq \$$
do
  if $\neg$ *expanded(S)* **then** *specialized(S)* := **false** ; $A := EXPAND(E, A, S)$ **fi**;
  if $\neg$ *specialized(S)* **then** $A := SPECIALIZE(E, A, S, M)$ **fi**;
  $S := A.SelTrans(S, a_i)$
  $i := i + 1$
od
return $(M\text{-}FINAL(S, M), A)$

In the modular case, a state is accepting if one of its positions corresponds to the end marker \$ and that position is part of the current selection. This is defined by the following algorithm.

Algorithm *M-FINAL*
Determine whether a given state is an accepting state in a given selection of modules.
Input. A state S and a list of modules M.
Output. **true** or **false**
Method.
  return $\exists\, p \in S\ [symbol(p) = \$ \wedge module(p) \in M]$

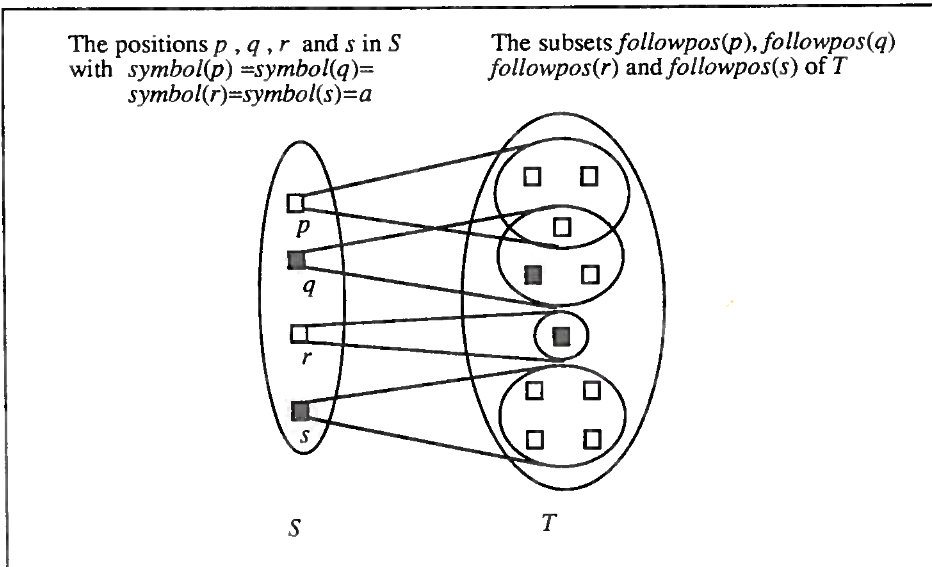


Figure 10. Positions causing a transition between states S and T on symbol a in the modular case.

The specialization of PDFA A for selection M (denoted by A/M) described by *SelTrans* has three interesting properties:

- For each string in the language defined by the restricted set of regular expressions E/M, the accepting sequence of positions in A/M is identical to the accepting sequence as it would occur in the new automaton A' that is constructed independently for the restricted set of regular expressions E/M.
- The above mentioned automaton A' does *not* need to occur as sub-automaton of A/M, since two distinct states S and T in A/M may become effectively equivalent due to specialization (i.e., after specialization S and T contain the same subset of positions that are labelled with a module name in the current selection and will thus behave identically; they correspond to a single state in automaton A'), but they will remain distinct states in A/M.
- It is difficult to assess the complexity of the operations *SPECIALIZE* and *EXPAND*. The latter constructs unions of sets (of positions) and has to perform a complex membership test to determine whether a newly constructed state already exists, while the former does no set construction at all but only performs pair-wise comparisons of set elements. Assuming that set construction is the most expensive operation, we conjecture that *SPECIALIZE* is cheaper than *EXPAND*.

3.3. Modular regular grammars

In the previous section we have discussed lexical definitions that have the form of a list of modular regular expressions. Each position occurring in these expressions is labelled with both a position name and a name of a module. Now we turn our attention to the complete modular regular grammars as sketched in Section 1. Such a grammar consists of two parts: abbreviations and rules. Both abbreviations and rules contain triples of the form

module-name: *token-name* = *regular-expression*.

Names appearing in a regular expression should always have been defined by a previous abbreviation or rule and can always be eliminated by textual substitution. When several regular expressions e_i are associated with one name, we associate with that name a regular expression containing all expressions e_i as alternatives. We will now show how a modular regular grammar of this form can be reduced to a set of modular regular expressions as defined in Section 3.1. We proceed in four stages:

1. Associate module names and positions with all alphabet symbols in the modular regular grammar.
2. Replace all uses of names in regular expressions by their definition.
3. Terminate all resulting expressions in the rules section with a $ symbol and associate both the module name preceding the expression and a new position with that $ symbol. When this terminator appears in a state (= set of positions) it will uniquely identify this rule. This fact can be used to determine the token-name to be associated with the recognized input string.
4. The set of modular regular expressions obtained in step 3 is the reduced form of the original modular regular grammar.

Only step 2 is non-trivial and requires some further comments, since what will happen when a named expression that is *not* selected is used in a expression that *is* selected? Intuitively, one would like to replace the use of the named expression by $\perp$. It turns out that this can be achieved by an appropriate definition of textual substitution.

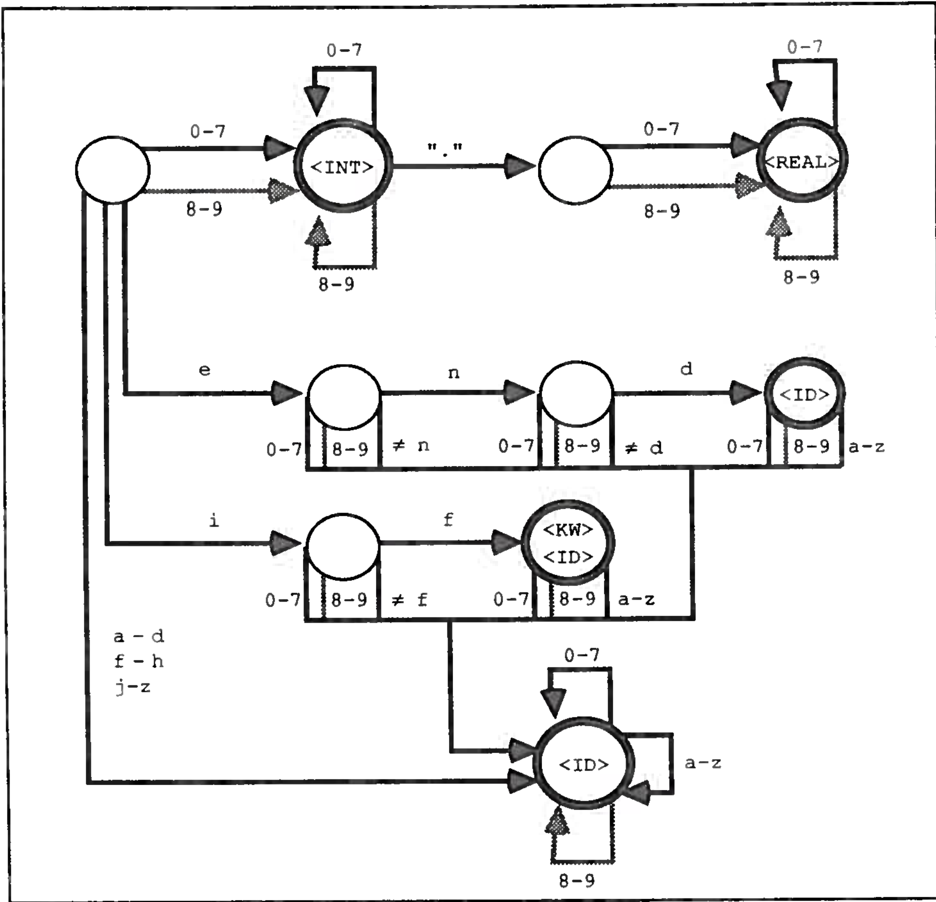


Figure 11. DFA corresponding to the selection { M1, M3, M4, M5, M6, M7 }.

Let e be the regular expression associated with some name in a modular, regular grammar E. Define $e' = copy(e)$ as the labelled regular expression obtained by taking a literal copy of e, with the exception that each symbol a_p appearing in e is replaced by $a_{p'}$, where p' is a new, unique, position label and we define $module(p')$=$module(p)$. It is important to note that the module name associated with each position remains the same.

Using this definition of taking a copy of a regular expression, all named regular expressions can be removed from a grammar by replacing each occurrence of a name by a copy of its associated expression. The positions occurring in the resulting, expanded, regular expression may now be labelled with different module names (this possibility was already mentioned in Section 3.1).

We conclude this section by applying all the techniques described so far to the modular regular grammar given as example in Figure 1 (see Section 1). In Figure 11, the complete DFA corresponding to this grammar is show for the selection of modules { M1, M3, M4,

M5, M6, M7 }, in others words the modules M2 (that includes the digits 8 and 9 in the definition of <DIGIT> and M8 (the keyword end) are not selected. The following conventions have been used in this figure:

- Invalid transitions are (as before) indicated by shaded arrows.
- Potentially accepting states are labelled with the name of the accepting token.
- The abbreviation $\neq c$ stands for all letters a-z, except the letter c.

Note that all transitions labelled with 8-9 are invalid and that the state that could potentially recognize end both as a keyword and as an identifier can only recognize it as an identifier in this particular selection.

4. CONCLUDING REMARKS

In this paper we have presented the algorithms required for the generation of lexical scanners for modular regular grammars. A prototype implementation of these algorithms has been completed and indicates that the method of selecting a sub-automaton from the large automaton corresponding to *all* regular expressions in *all* modules is superior over the method of constructing a new automaton for each selection of modules. It is too early to present a quantitative analysis of the performance of this implementation.

As indicated in Section 1, the modularization of language definitions implies that all parts of such a definition have to be processed in a modular fashion. In [Rek89], a technique is sketched for the generation of parsers for modular context-free grammars. It turns out that the techniques for lazy and incremental program generation as described earlier in [HKR87a], form a good foundation for modular program generation techniques.

ACKNOWLEDGEMENTS

Jan Rekers made several comments on a draft of this paper. The technique presented here was inspired by our discussions on modular parser generation. All these methods are extensions of the lazy/incremental program generation techniques developed in cooperation with Jan Heering and Jan Rekers.

REFERENCES

[BS87] G. Berry & R. Sethi, "From regular expressions to deterministic automata", INRIA Report 649, 1987.

[HKR87a] J. Heering, P. Klint & J. Rekers, "Principles of lazy and incremental program generation", Centre for Mathematics and Computer Science, Report CS-R8749.

[HKR87b] J. Heering, P. Klint & J. Rekers, "Incremental generation of lexical scanners", Centre for Mathematics and Computer Science, Report CS-R8761.

[MY60] R. McNaughton & H. Yamada, "Regular expressions and state graphs for automata", *IRE Transactions on Electronic Computers*, **EC-9** (1960), pp. 38-47.

[Rek89] J. Rekers, "Modular parser generation", manuscript, 1989.

Metric Semantics for the Input/Output Behaviour of Sequential Programs

Joost N. Kok
University of Utrecht*

Abstract

As in the book *Mathematical Theory of Program Correctness* by J.W. de Bakker we assign both operational and denotational semantics to a language with while and one with simultaneous recursion. The main difference is that we use tools from metric topology instead of order theory.

1 Introduction

The book *Mathematical Theory of Program Correctness* by J.W. de Bakker ([dB80]) has become a standard work in the field of the semantics and correctness of sequential programming. The mathematical basis of this book is order theory. An important role is played by the following fixed point theorem: continuous functions on a complete partial ordering have least fixed points. This theorem is applied in fixed point definitions: for example in the definition of a semantic model for a language with recursive procedures. This is a typical situation in which functions have more than one fixed point.

In later work of J.W. de Bakker metric topology is used as the main mathematical tool for semantic theories, for example in [dBZ82] or [dBM88]. The metric equivalent of the fixed point theorem for continuous functions is Banach's theorem: a contraction on a complete metric space has a unique fixed point. This theorem is works very well in situations where functions have have unique fixed points.

Both fixed point theorems state that the (least) fixed point can be obtained by an iteration procedure. in the first case as the least upperbound of a chain starting in the least element of the complete partial ordering, in the second case as the limit of a Cauchy sequence starting from an arbitrary element of the metric space. In both cases each next element is obtained by applying the funtion to the previous element.

It has been an open question whether we can apply metric theory in non unique fixed point situations. In this paper we take as a starting point chapters three and four of *Mathematical Theory of Program Correctness* and show how we can treat the semantic theory also in a metric way. The first chapter deals with a language containing

*Department of Computer Science, University of Utrecht, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands.

assignment, sequential composition, if statement and while statement. The second chapter gives semantics to a language in which in addition we have simultaneous recursion. Our goal is to give to both languages input/output semantics (a function from the initial state to (if it exists) the final state): an operational one by using transition systems and denotational with the aid of metric topology. In both cases we derive two independent characterizations of the denotational semantics:

1. it is the limit of a Cauchy sequence,
2. it is the fixed point with the least distance to the nowhere defined function.

We also study the equivelence between the different semantic models. It turns out that the proofs of the equivalences are a mixture of the proofs in *Mathematical Theory of Program Correctness* and [KR88].

2 While statements

We start by giving the set of statements. The notation $(x \in)X$ introduces the set X with typical element x ranging over X. The set of integers and the set of booleans are denoted by $\mathbb{N}$ and $\mathbb{B}$ respectively.

Definition 2.1 *Let $(b \in)$ BExp, $(x, y \in)$ Var, and $(s \in)$ Exp be sets. Let $(S \in)$ Stat be the set statements specified by*

$$S \ ::= \ x := s \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } S \textbf{ od}.$$

Next we introduce the notion of a state:

Definition 2.2 *The set Σ of states is given by $\Sigma = Var \to \mathbb{N}$.*

We use the notation $\sigma[x := \alpha]$, with $\alpha \in \mathbb{N}$, for a variant of σ, i.e., for the state which is defined by

$$\sigma[x := \alpha] = \lambda y. \begin{cases} \alpha & \textit{if } x = y \\ \sigma(y) & \textit{otherwise} \end{cases}$$

We assume given functions $\mathcal{V} : Exp \to \Sigma \to \mathbb{N}$ and $\mathcal{W} : BExp \to \Sigma \to \mathbb{B}$. We now give the definition of the operational semantics. It is based on a transition system. Here, a transition is a fourtuple in $Stat \times \Sigma \times (Stat \cup \{E\}) \times \Sigma$, written in the notation

$$(S, \sigma) \to (S', \sigma')$$

or

$$(S, \sigma) \to (E, \sigma').$$

The symbol E is called the empty statement and stands for termination. We present a formal transition system T which consists of axioms (in one of the two forms above) or rules, in the form

$$\frac{(S_1, \sigma_1) \to (S_1', \sigma_1')}{(S_2, \sigma_2) \to (S_2', \sigma_2')}$$

in which both S_1' and S_2' can be replaced by E. Transitions which are given as axioms hold by definition. Moreover, a transition which is the consequence of a rule holds whenever it can be established that its premise holds.

Axioms:

$$(x := s, \sigma) \to (E, \sigma[x := \mathcal{V}(s)(\sigma)])$$

$$(\textbf{while } b \textbf{ do } S \textbf{ od}, \sigma) \to (E, \sigma) \textit{ if } \mathcal{W}(b)(\sigma) = \textit{ff}$$

Rules:

$$\begin{array}{l} \dfrac{(S_1, \sigma) \to (S_1', \sigma')}{(S_1; S, \sigma) \to (S_1'; S, \sigma')} \\ (\textbf{while } b \textbf{ do } S_1 \textbf{ od}, \sigma) \to (S_1'; \textbf{while } b \textbf{ do } S_1 \textbf{ od}, \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = \textit{tt} \\ (\textbf{if } b \textbf{ then } S_1 \textbf{ else } S \textbf{ fi}, \sigma) \to (S_1', \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = \textit{tt} \\ (\textbf{if } b \textbf{ then } S \textbf{ else } S_1 \textbf{ fi}, \sigma) \to (S_1', \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = \textit{ff} \end{array}$$

$$\begin{array}{l} \dfrac{(S_1, \sigma) \to (E, \sigma')}{(S_1; S, \sigma) \to (S, \sigma')} \\ (\textbf{while } b \textbf{ do } S_1 \textbf{ od}, \sigma) \to (\textbf{while } b \textbf{ do } S_1 \textbf{ od}, \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = \textit{tt} \\ (\textbf{if } b \textbf{ then } S_1 \textbf{ else } S \textbf{ fi}, \sigma) \to (E, \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = \textit{tt} \\ (\textbf{if } b \textbf{ then } S \textbf{ else } S_1 \textbf{ fi}, \sigma) \to (E, \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = \textit{ff} \end{array}$$

Let $\Sigma_\perp = \Sigma \cup \{\perp\}$, where $\perp$ is a special. From now on, σ ranges over $\Sigma_\perp$. We proceed with the definition of the operational semantics $\mathcal{O}$.

Definition 2.3 *The mapping $\mathcal{O} : Stat \to \Sigma_\perp \to \Sigma_\perp$ is given by*

$$\mathcal{O}(S)(\sigma) = \begin{cases} \perp & \sigma = \perp \\ \sigma' & (S, \sigma) \to^* (E, \sigma') \\ \perp & \textit{otherwise} \end{cases}$$

The next step is the development of a metric denotational model. We have to assume that the set $\Sigma_\perp$ is countable, i.e. $\Sigma_\perp = \{\sigma_1, \sigma_2, \ldots\}$. A metric d on $\Sigma_\perp \to \Sigma_\perp$ is defined as follows: Let $f, g \in \Sigma_\perp \to \Sigma_\perp$.

$$d(f, g) = \sum_{i=1}^{\infty} \begin{cases} 0 & f(\sigma_i) = g(\sigma_i) \\ 10^{-i} & f(\sigma_i) \neq g(\sigma_i). \end{cases}$$

The denotational semantics $\mathcal{D} : Stat \to \Sigma_\perp \to \Sigma_\perp$ is given in

Definition 2.4

1. $\mathcal{D}(x := s) = \lambda\sigma. \begin{cases} \bot & \sigma = \bot \\ \sigma[x := \mathcal{V}(s)(\sigma)] & \textit{otherwise} \end{cases}$

2. $\mathcal{D}(S_1; S_2) = \mathcal{D}(S_2) \circ \mathcal{D}(S_1)$

3. $\mathcal{D}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \lambda\sigma. \begin{cases} \bot & \sigma = \bot \\ \mathcal{D}(S_1)(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \mathcal{D}(S_2)(\sigma) & \mathcal{W}(b)(\sigma) = ff \end{cases}$

4. $\mathcal{D}(\textbf{while } b \textbf{ do } S \textbf{ od}) = \lim_{i\to\infty} \phi_i$ *where*

$$\phi_0 = \lambda\sigma.\bot$$

$$\phi_{i+1} = \lambda\sigma. \begin{cases} \bot & \sigma = \bot \\ (\phi_i \circ \mathcal{D}(S))(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \sigma & \mathcal{W}(b)(\sigma) = ff \end{cases}$$

Lemma 2.5 *The sequence $(\phi_i)_i$ is a Cauchy sequence in $\Sigma_\bot \to \Sigma_\bot$.*

Proof: Follows directly from $\forall i[\phi_i(\sigma) = \sigma' \wedge \sigma' \neq \bot \Rightarrow \phi_{i+1}(\sigma) = \sigma']$. □

Theorem 2.6 *Let $\Psi : (\Sigma_\bot \to \Sigma_\bot) \to (\Sigma_\bot \to \Sigma_\bot)$ be defined as follows.*

$$\Psi(F)(\sigma) = \lambda\sigma. \begin{cases} \bot & \sigma = \bot \\ (F \circ \mathcal{D}(S))(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \sigma & \mathcal{W}(b)(\sigma) = ff. \end{cases}$$

Let $\phi = \mathcal{D}(\textbf{while } b \textbf{ do } S \textbf{ od})$. Then ϕ is the fixed point of Ψ with the smallest distance to $\lambda\sigma.\bot$.

We continue with the derivation of the equivalence $\mathcal{O} = \mathcal{D}$. The mapping $\Phi : (Stat \to (\Sigma_\bot \to \Sigma_\bot)) \to (Stat \to (\Sigma_\bot \to \Sigma_\bot))$ is given in

Definition 2.7
$$\Phi(F)(S)(\sigma) = \begin{cases} \bot & \sigma = \bot \\ \sigma' & (S,\sigma) \to (E,\sigma') \\ F(S')(\sigma') & (S,\sigma) \to (S',\sigma') \end{cases}$$

An important step in the proof that $\mathcal{O} = \mathcal{D}$ holds on *Stat* is the following lemma:

Lemma 2.8

1. $\Phi(\mathcal{D})(S_1; S_2) = \mathcal{D}(S_2) \circ \Phi(\mathcal{D})(S_1)$

2. $\Phi(\mathcal{D})(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \lambda\sigma. \begin{cases} \bot & \sigma = \bot \\ \Phi(\mathcal{D})(S_1)(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \Phi(\mathcal{D})(S_2)(\sigma) & \mathcal{W}(b)(\sigma) = ff \end{cases}$

3. $\Phi(\mathcal{D})(\textbf{while } b \textbf{ do } S \textbf{ od}) = \lambda\sigma. \begin{cases} \bot & \sigma = \bot \\ (\mathcal{D}(\textbf{while } b \textbf{ do } S \textbf{ od}) \circ \Phi(\mathcal{D})(S))(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \sigma & \mathcal{W}(b)(\sigma) = ff. \end{cases}$

We have the following

Corollary 2.9 *$\mathcal{D}$ is a fixed point of Φ.*

The equivalence $\mathcal{O} = \mathcal{D}$ follows by the following

Theorem 2.10 *For all $S \in Stat$, $\sigma, \sigma' \in \Sigma$*

$$(S,\sigma) \to^* (E,\sigma') \Leftrightarrow \mathcal{D}(S)(\sigma) = \sigma' \wedge \sigma' \neq \bot$$

Proof:

$\Rightarrow$:

Assume $(S,\sigma) \to^n (E,\sigma')$. Using the corollary we have

$$\mathcal{D} = \Phi(\mathcal{D}) = \cdots = \Phi^n(\mathcal{D}) =$$

$$\lambda\sigma.\begin{cases} \bot & \sigma = \bot \\ \sigma' & (S,\sigma) \to^k (E,\sigma') \wedge k \leq n \\ \mathcal{D}(S')(\sigma') & (S,\sigma) \to^n (E,\sigma'). \end{cases}$$

Hence $\mathcal{D}(S)(\sigma) = \sigma'$.

$\Leftarrow$:

We prove the result by induction on the complexity of the statement S. We distinguish the following cases:

1. $S \equiv x := s$:

$$\mathcal{D}(S)(\sigma) = \sigma' \wedge \sigma' \neq \bot \Rightarrow \sigma' = \sigma[x := \mathcal{V}(s)(\sigma)]$$

   and

$$(x := s, \sigma) \to (E, \sigma[x := \mathcal{V}(s)(\sigma)])$$

2. $S \equiv S_1; S_2$:

$$\mathcal{D}(S_1;S_2)(\sigma) = \sigma' \wedge \sigma' \neq \bot \Rightarrow$$

$$\exists\sigma''[\sigma'' = \mathcal{D}(S_1)(\sigma) \wedge \sigma' = \mathcal{D}(S_2)(\sigma'')] \Rightarrow$$

$$\exists\sigma''[(S_1,\sigma) \to^* (E,\sigma'') \wedge (S_2,\sigma'') \to^* (E,\sigma')] \Rightarrow$$

$$(S_1;S_2,\sigma) \to^* (E,\sigma')$$

3. $S \equiv$ **while** b **do** S_1 **od**:

$$\mathcal{D}(\textbf{while } b \textbf{ do } S_1 \textbf{ od})(\sigma) = \sigma' \wedge \sigma' \neq \bot \Rightarrow$$

$$(\lim_{i\to\infty}\phi_i)(\sigma) = \sigma' \wedge \sigma' \neq \bot \Rightarrow$$

$$\exists k[(\phi_k)(\sigma) = \sigma' \wedge \sigma' \neq \bot] \Rightarrow$$

$$\exists k[\sigma' = \underbrace{\mathcal{D}(S_1)\circ\cdots\mathcal{D}(S_1)}_{k}(\sigma) \wedge \mathcal{W}(b)(\sigma') = ff \wedge$$

$$\forall j : j < k[\mathcal{W}(b)\underbrace{\mathcal{D}(S_1)\circ\cdots\mathcal{D}(S_1)}_{j}(\sigma) = tt] \wedge \sigma' \neq \bot] \Rightarrow$$

$$\exists\sigma_1,\ldots,\sigma_{k+1}[$$

$$\sigma_1 = \sigma \wedge \sigma_2 = \mathcal{D}(S_1)(\sigma_1)\cdots \wedge \sigma_{k+1} = \mathcal{D}(S_1)(\sigma_k) \wedge \sigma_{k+1} = \sigma' \wedge$$

$$\mathcal{W}(b)(\sigma') = ff \wedge \forall i \in \{1,\ldots,k\}[\mathcal{W}(b)(\sigma_i) = tt]] \Rightarrow$$

$$\exists\sigma_1,\ldots,\sigma_{k+1}[$$

$$(S_1,\sigma_1) \to^* (E,\sigma_2) \wedge \mathcal{W}(b)(\sigma_1) = tt \wedge$$

$$(S_1,\sigma_2) \to^* (E,\sigma_3) \wedge \mathcal{W}(b)(\sigma_2) = tt \wedge$$

$$\ldots$$

$$(S_1,\sigma_k) \to^* (E,\sigma_{k+1}) \wedge \mathcal{W}(b)(\sigma_k) = tt \wedge$$

$$\mathcal{W}(b)(\sigma_{k+1}) = ff] \Rightarrow$$

$$(\textbf{while } b \textbf{ do } S_1 \textbf{ od}, \sigma) \to^* (E,\sigma').$$

□

3 Recursion

We now present the syntax for a language with recursion. Again, we use S to range over the set of statements *Stat*. The set of statements is extended with a new syntactic category, the set $(P \in)PVar$ of procedure variables.

Definition 3.1

1. *Let* $(S \in)$ *Stat be the set statements specified by*

$$S \ ::= \ x := s \mid S_1;S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid P.$$

2. *The set* ($G \in$) *GStat of guarded statements is given by*

$$G ::= x := s \mid G_1; S_2 \mid \textbf{if } b \textbf{ then } G_1 \textbf{ else } G_2 \textbf{ fi}$$

3. *The set* ($D \in$) *Decl of declarations consists of* n*-tuples* $D \equiv P_1 \Leftarrow G_1, \ldots, P_n \Leftarrow G_n$.

4. *The set* ($R \in$) *Prog of programs consists of tuples* $R \equiv D \mid S$.

We use simultaneous recursion rather than using so-called μ-constructs. In procedure declarations we only use guarded statements. The set of guarded statements is a subset of the set of statements. Declarations with unguarded statements can be made guarded with *skip*-statements. For the input/output behaviour this does not make any difference.

Now, a transition is a fourtuple in $Prog \times \Sigma \times (Prog \cup \{E\}) \times \Sigma$, written in the notation

$$(R, \sigma) \rightarrow (R', \sigma')$$

or

$$(R, \sigma) \rightarrow (E, \sigma').$$

The transition system which generates the transition relation is given in

Axioms:

$$(x := s, \sigma) \rightarrow (E, \sigma[x := \mathcal{V}(s)(\sigma)])$$

Rules:

$$\frac{(D \mid S_1, \sigma) \rightarrow (D \mid S_1', \sigma')}{(D \mid S_1; S, \sigma) \rightarrow (D \mid S_1'; S, \sigma')}$$

$$(D \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S \textbf{ fi}, \sigma) \rightarrow (D \mid S_1', \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = tt$$

$$(D \mid \textbf{if } b \textbf{ then } S \textbf{ else } S_1 \textbf{ fi}, \sigma) \rightarrow (D \mid S_1', \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = \mathit{ff}$$

$$(\ldots, P \Leftarrow S_1, \ldots \mid P, \sigma) \rightarrow (\ldots, P \Leftarrow S_1, \ldots \mid S_1', \sigma')$$

$$\frac{(D \mid S_1, \sigma) \rightarrow (E, \sigma')}{(D \mid S_1; S, \sigma) \rightarrow (D \mid S, \sigma')}$$

$$(D \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S \textbf{ fi}, \sigma) \rightarrow (E, \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = tt$$

$$(D \mid \textbf{if } b \textbf{ then } S \textbf{ else } S_1 \textbf{ fi}, \sigma) \rightarrow (E, \sigma') \textit{ if } \mathcal{W}(b)(\sigma) = \mathit{ff}$$

$$(\ldots, P \Leftarrow S_1, \ldots \mid P, \sigma) \rightarrow (E, \sigma')$$

We next define the operational semantics.

Definition 3.2 *The mapping* $\mathcal{O} : Prog \rightarrow \Sigma_\perp \rightarrow \Sigma_\perp$ *is given by*

$$\mathcal{O}(R)(\sigma) = \begin{cases} \perp & \sigma = \perp \\ \sigma' & (R, \sigma) \rightarrow^* (E, \sigma') \\ \perp & \textit{otherwise} \end{cases}$$

We introduce the notion of environment which is used to store and retrieve meanings of procedure variables. Let $\Gamma = PVar \to \Sigma_\perp \to \Sigma_\perp$ be the set of environments, and let $\gamma \in \Gamma$. We write $\gamma[P_i := \phi_i]$ for a variant of γ which is like γ but with $\gamma(P_i) = \phi_i$. We are now in a position to define the denotational semantics for $R \in Prog$. The denotational semantics $\mathcal{D} : \Gamma \to Stat \to \Sigma_\perp \to \Sigma_\perp$ is given in

Definition 3.3

1. $\mathcal{D}(\gamma)(x := s) = \lambda\sigma. \begin{cases} \perp & \sigma = \perp \\ \sigma[x := \mathcal{V}(s)(\sigma)] & \textit{otherwise} \end{cases}$

2. $\mathcal{D}(\gamma)(S_1; S_2) = \mathcal{D}(\gamma)(S_2) \circ \mathcal{D}(\gamma)(S_1)$

3. $\mathcal{D}(\gamma)(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \lambda\sigma. \begin{cases} \perp & \sigma = \perp \\ \mathcal{D}(\gamma)(S_1)(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \mathcal{D}(\gamma)(S_2)(\sigma) & \mathcal{W}(b)(\sigma) = ff \end{cases}$

4. $\mathcal{D}(\gamma)(P) = \gamma(P)$

We define a metric d on Γ by putting

$$d(\gamma_1, \gamma_2) = \sup\{d(\gamma_1(P), \gamma_2(P)) : P \in PVar\}.$$

This metric turns Γ into a complete metric space.

Choose an arbitrary declaration $D \equiv P_1 \Leftarrow G_1, \ldots, P_n \Leftarrow G_n$. Let the mapping $\Psi : \Gamma \to \Gamma$ be given by

$$\Psi(\gamma) = \gamma[P_i := \mathcal{D}(\gamma)(G_i)].$$

We have the following lemma.

Lemma 3.4 *The sequence $(\Psi^j(\lambda P.\lambda\sigma.\perp))_j$ is a Cauchy sequence.*

Proof Follows from

$$\forall j[\Psi^j(\lambda P.\lambda\sigma.\perp)(P)(\sigma) = \sigma' \wedge \sigma' \neq \perp \Rightarrow \Psi^{j+1}(\lambda P.\lambda\sigma.\perp)(P)(\sigma) = \sigma'].$$

□

Theorem 3.5 *Let $\gamma_D = \lim_{i\to\infty} \Psi^i(\lambda P.\lambda\sigma.\perp)$. Then γ_D is the fixed point of Ψ with the least distance to $\lambda P.\lambda\sigma.\perp$.*

The mapping $\mathcal{M} : Prog \to \Sigma_\perp \to \Sigma_\perp$ is given as follows:

$$\mathcal{M}(D \mid S) = \mathcal{D}(\gamma_D)(S).$$

We turn to the equivalence $\mathcal{O} = \mathcal{M}$. The mapping $\Phi : (Prog \to (\Sigma_\perp \to \Sigma_\perp)) \to (Prog \to (\Sigma_\perp \to \Sigma_\perp))$ is given in

Definition 3.6

$$\Phi(F)(R)(\sigma) = \begin{cases} \bot & \sigma = \bot \\ \sigma' & (R,\sigma) \rightarrow (E,\sigma') \\ F(R')(\sigma') & (R,\sigma) \rightarrow (R',\sigma') \end{cases}$$

An important step in the proof that $\mathcal{O} = \mathcal{M}$ holds on *Prog* is the following lemma:

Lemma 3.7

1. $\Phi(\mathcal{M})(D \mid S_1; S_2) = \mathcal{M}(D \mid S_2) \circ \Phi(\mathcal{M})(D \mid S_1)$
2. $\Phi(\mathcal{M})(\ldots, P \Leftarrow G, \ldots \mid P) = \Phi(\mathcal{M})(\ldots, P \Leftarrow G, \ldots \mid G)$
3. $\Phi(\mathcal{M})(D \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \lambda\sigma. \begin{cases} \bot & \sigma = \bot \\ \Phi(\mathcal{M})(D \mid S_1)(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \Phi(\mathcal{M})(D \mid S_2)(\sigma) & \mathcal{W}(b)(\sigma) = ff \end{cases}$

We have the following

Corollary 3.8 *$\mathcal{M}$ is a fixed point of Ψ.*

Proof: We show for any $D \in Decl$ and $S \in Stat$

$(*)\ \Phi(\mathcal{M})(D \mid S) = \mathcal{M}(D \mid S)$

The proof proceeds in two stages; first we show $(*)$ for $G \in GStat (\subseteq Stat)$ and next for $S \in Stat$. We prove the result for $G \in GStat (S \in Stat)$ by induction on the complexity of the statement $G(S)$.

1. We only treat the cases $G \equiv G_1; S$ and $G \equiv \textbf{if } b \textbf{ then } G_1 \textbf{ else } G_2 \textbf{ fi}$.

   $G \equiv G_1; S$:

   $\Phi(\mathcal{M})(D \mid G_1; S) = [lemma]$

   $\mathcal{M}(D \mid S) \circ \Phi(\mathcal{M})(D \mid G_1) = [induction]$

   $\mathcal{M}(D \mid S) \circ \mathcal{M}(D \mid G_1) =$

   $\mathcal{M}(D \mid G_1; S)$.

   $G \equiv \textbf{if } b \textbf{ then } G_1 \textbf{ else } G_2 \textbf{ fi}$:

   $\Phi(\mathcal{M})(D \mid \textbf{if } b \textbf{ then } G_1 \textbf{ else } G_2 \textbf{ fi}) = [lemma]$

   $$\lambda\sigma. \begin{cases} \bot & \sigma = \bot \\ \Phi(\mathcal{M})(D \mid G_1)(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \Phi(\mathcal{M})(D \mid G_2)(\sigma) & \mathcal{W}(b)(\sigma) = ff \end{cases} = [induction]$$

$$\lambda\sigma.\begin{cases} \perp & \sigma = \perp \\ \mathcal{M}(D \mid G_1)(\sigma) & \mathcal{W}(b)(\sigma) = tt \\ \mathcal{M}(D \mid G_2)(\sigma) & \mathcal{W}(b)(\sigma) = ff \end{cases} =$$

$\mathcal{M}(D \mid \textbf{if } b \textbf{ then } G_1 \textbf{ else } G_2 \textbf{ fi})$.

2. We only treat the case

   $S \equiv P$:

$$\Phi(\mathcal{M})(\ldots, P \Leftarrow G, \ldots \mid P) = [lemma]$$

$$\Phi(\mathcal{M})(\ldots, P \Leftarrow G, \ldots \mid G) = [G \in GStat]$$

$$\mathcal{M}(\ldots, P \Leftarrow G, \ldots \mid G) = [induction]$$

$$\mathcal{M}(\ldots, P \Leftarrow G, \ldots \mid P).$$

□

Theorem 3.9 $\quad (R, \sigma) \rightarrow^* (E, \sigma') \Leftrightarrow \mathcal{M}(R)(\sigma) = \sigma' \wedge \sigma' \neq \perp$.

Proof:

$\Rightarrow$:

Assume $(R, \sigma) \rightarrow^n (E, \sigma')$. Using the corollary we have

$$\mathcal{M} = \Phi(\mathcal{M}) = \cdots = \Phi^n(\mathcal{M}) =$$

$$\lambda\sigma.\begin{cases} \perp & \sigma = \perp \\ \sigma' & (R, \sigma) \rightarrow^k (E, \sigma') \wedge k \leq n \\ \mathcal{D}(R')(\sigma') & (R, \sigma) \rightarrow^n (E, \sigma'). \end{cases}$$

Hence $\mathcal{M}(S)(\sigma) = \sigma'$.

$\Leftarrow$: Let, for each k, $\gamma_k = \Psi^k(\lambda P.\lambda\sigma.\perp)$. Note that $\lim_{k\rightarrow\infty} \gamma_k = \gamma_D$. We prove for all k, S, σ, σ'

$$\sigma' = \mathcal{D}(\gamma_k)(S)(\sigma) \wedge \sigma' \neq \perp \Rightarrow (D \mid S, \sigma) \rightarrow^* (E, \sigma').$$

We prove it by induction on the tuples made up of k and the complexity of the statement S. As the ordering on such tuples we take the lexicographical ordering. We only treat the cases $S \equiv S_1; S_2$ and $S \equiv P$.

$S \equiv S_1; S_2$:

$$\mathcal{D}(\gamma_k)(S_1; S_2) =$$

$$\mathcal{D}(\gamma_k)(S_2) \circ \mathcal{D}(\gamma_k)(S_1).$$

Because

$$\sigma' = \mathcal{D}(\gamma_k)(S_1; S_2)(\sigma) \wedge \sigma' \neq \bot$$

we can find a σ'' such that

$$\sigma'' = \mathcal{D}(\gamma_k)(S_1)(\sigma) \wedge \sigma'' \neq \bot$$

$$\sigma' = \mathcal{D}(\gamma_k)(S_2)(\sigma'') \wedge \sigma' \neq \bot.$$

By induction

$$(D \mid S_1, \sigma) \rightarrow^* (E, \sigma'')$$

$$(D \mid S_2, \sigma'') \rightarrow^* (E, \sigma')$$

and hence

$$(D \mid S_1; S_2, \sigma'') \rightarrow^* (E, \sigma').$$

$S \equiv P$: Let G the body of P.

$$\mathcal{D}(\gamma_k)(P) =$$

$$\gamma_k(P) =$$

$$\mathcal{D}(\gamma_{k_1})(G).$$

Hence by induction

$$(D \mid G, \sigma) \rightarrow^* (E, \sigma')$$

and this implies

$$(D \mid P, \sigma) \rightarrow^* (E, \sigma').$$

□

4 Conclusion

We hope that this paper shows that we also can handle the input/output semantics of sequential languages with the help of metric topology. In this case, the difference between using order theory or metric topology is not very big. The proofs are in both cases very similar.

References

[dB80] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice/Hall, 1980.

[dBM88] J.W. de Bakker and J.-J.Ch. Meyer. Metric semantics for concurrency. *BIT*, 28:504–529, 1988.

[dBZ82] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Inform. and Control*, 54:70–120, 1982.

[KR88] J.N. Kok and J.J.M.M. Rutten. Contractions in comparing concurrency semantics. In T. Lepistö and A. Salomaa, editors, *Proc. 15th International Colloquium Automata, Languages and Programming*, pages 317–332, Springer Verlag, 1988. Lecture Notes in Computer Science 317.

CORRECTNESS OF THE TWO-PHASE COMMIT PROTOCOL

Jan van Leeuwen

Abstract

The 2-phase commit protocol is a standard algorithm for safeguarding the atomicity of transactions in a distributed system. A self-contained description of the 2-phase commit protocol is presented and verified.

Keywords and phrases: distributed systems, client/server model, message passing, atomic actions, two-phase commit protocol, correctness criteria.

1 Introduction

In order to deal with the complex issues of communication and control in a distributed system, it is necessary to have a consistent architectural model underlying the design and development of a system. In many distributed systems the client/server paradigm is used to explain the underlying system view, suggesting the possibility of a formal model and correctness proofs of the design. The rationale for the Client/Server (or Customer/Server) model was first described by Gentleman[3], who identified the issues that need to be resolved in any system designed from this perspective. The model is heavily based on the "abstract object" approach to distributed system design (Watson [8]).

The Client/Server model can be viewed as a unifying framework for the high level description and specification of communications and interactions between processes. One or more processes, identified as the clients, request a service from some other process, which acts as the server. The server, after performing the requested service, posts replies to the clients. The client-server relationships exist for the duration of the interaction. For a detailed study of the Client/Server model, see van Leeuwen [7].

The main topic of this note concerns the more involved "atomic" interactions between clients and servers and the need for atomic commit protocols. An integral development and correctness proof is given for a standard 2-phase commit protocol that seems to be appreciably simpler and more clarifying than similar treatments in the current literature (see e.g. [1, 2]).

1.1 Managing Atomic Actions

The Client/Server model assumes that clients and servers interact strictly on a request/reply basis. In applications it may be desirable that a client and a server interact in a more complex manner during a session and engage in an activity (a set of operations or "an action") that affects the information stored at the client and the server simultaneously and in an indivisible manner. Activities of this kind are called "transactions" or "atomic

actions". Atomic actions would pose no particular problem if it weren't for the fact that in all realistic applications "exceptions" and "failures" (like link failures or site crashes) can occur during an atomic action. We will use the term "failure" to refer to any abnormal condition that arises during an atomic action. The possibility of failures requires that the effects of an atomic action must be recoverable at all times throughout its elaboration, until the atomic action can be regarded as "safely completed" at both ends. The implementation of atomic actions thus requires two basic facilities (see e.g. Gray [4]):

(i) a recovery mechanism, i.e., a mechanism for "undo-ing" the effect of one or more atomic actions. Recovery mechanisms are always based on the use of logs that record information on the processing of atomic actions at each site, and on a method for effectuating a rollback. Logs must be recoverable in case of failures and thus must be kept on "stable storage".

(ii) an atomic commit protocol, i.e., a protocol for detecting "safe completion" and committing the effects of an atomic action at both ends. Atomic commit protocols are atomic actions and thus must be recoverable themselves.

The occurrence of a failure at some site does not necessarily imply that the atomic actions in progress must all be aborted and rolled back. It may be possible for a node to recover to a consistent state, based on information in its log. (This is called "independent recovery".) After recovery, a site may wish to have an atomic action re-started.

1.2 Implementation of atomic actions

The implementation of atomic actions in the context of possible failures is a well-studied problem. An excellent introduction to concurrency control and recovery in distributed databases was given by Bernstein et al. [2]. Also, a detailed recommendation for the implementation of atomic commit protocols appears in the CCR standard of ISO [5]. It suggests that atomic commit protocols follow some version of the well-known 2-phase commit protocol due to Gray [4] and Lampson & Sturgis [6], and use the following primitives:

(i) C - BEGIN

(ii) C - PREPARE

(iii) C - READY

(iv) C - REFUSE

(v) C - COMMIT

(vi) C - ROLLBACK

(vii) C - RESTART

Our main goal will be to give a more refined and complete presentation of the Individual Commit protocol than is usually given (cf. [1, 2]. As it will require no extra effort, we will describe the protocol for the more general case of a client-initiated atomic action that involves multiple servers. It is assumed that the client remains the "coordinator" (or "superior") of the atomic action and thus of the atomic commit protocol. The client

will only initiate the commit protocol (with a C-PREPARE primitive) if it has reason to do so, i.e., if the activity of the atomic action at its own site has ended (which implies that the servers have provided their operation results insofar as needed by the client). At all times during the atomic action, the client and the servers must be ready to honor a C-ROLLBACK or C-RESTART request from any party in the atomic action. Note that the servers only communicate with the client, but not with each other (during the atomic actions).

1.3 Atomic commit protocols

Applied to an atomic action, an atomic commit protocol is a distributed algorithm for the client and the servers that should guarantee that they all commit or all abort the atomic action. Following Bernstein et al. [1] the situation for an atomic commit protocol can be rephrased in more precise terms as follows. Each party (client or server) may cast exactly one of two votes: C-READY ("yes") or C-REFUSE ("no"), and can reach exactly one of two decisions: C-COMMIT ("commit") or C-ROLLBACK ("abort"). An atomic commit protocol must satisfy the following requirements:

AC1 : A party cannot change its vote after it has cast a vote.
AC2 : All parties that reach a decision reach the same decision.
AC3 : A party cannot change its decision after it has reached one.
AC4 : A C-COMMIT decision can be reached only if all parties voted and voted C-READY.
AC5 : If there are no failures and all parties voted C-READY, then the decision C-COMMIT will be reached by all parties.
AC6 : Consider any execution of the protocol in the context of permissible failures. At any point in this execution, if all current failures have been repaired and no new failures occur for a sufficiently long period of time, then all parties will eventually reach a decision.

The requirements can be viewed as the minimal correctness criteria for atomic commit protocols. (Except for AC1, the requirements are taken from Bernstein et al. [1].) AC1 through AC5 can usually be satisfied quite easily, but AC6 requires a suitable recovery procedure to be part of the protocol. Note that after voting C-READY, a party (client or server) can not be certain of what the decision will be until it has received sufficient information to decide. Until that moment, we say that the party is "uncertain". If a failure occurs that cuts an uncertain party off, then this party is said to be blocked. A blocked party cannot reach a decision until after the connection to the other parties has been restored. Blocking is usually concluded if no messages arrive during a certain timeout interval within the uncertainly period. (Heuristic commit protocols allow a blocked party to make a calculated guess of the decision. For example, a blocked client may be allowed to commit.) A different situation arises when a party fails (crashes) during its uncertainty period. In this case a more involved recovery procedure may have to be followed (see Bernstein et al. [1] or below).

2 The (2-phase) individual commit protocol

The standard (2-phase) atomic commit protocol is as follows, in terms of the recommended CCR primitives. (Note that the desired steps of the recovery procedure after failure are part of the overall protocol, but these are not included in the basic specification below.)

Individual Commit Protocol

Client's Commit

c-0. Vote ready;

Phase 1

c-1. **Write** "prepare" record **to** the Log;

c-2. **Send** C-PREPARE messages **to** all servers; **activate** timer;

c-3. {Await answer messages (C-READY or C-REFUSE) from all servers using a timer and act as follows}

Case condition **of**

c-3.1. Timeout or C-REFUSE message received:

begin

Write "rollback" record **to** the Log;

Send C-ROLLBACK messages **to** all servers;

C-ROLLBACK

end;

c-3.2. All servers answered and answered C-READY:

continue with Phase 2

end;

{End of Phase 1}

Phase 2

c-4. **Write** "commit" record **to** the Log;

c-5. **Send** C-COMMIT messages **to** all servers ; **activate** timer;

c-6. {Await answer messages (ACK) from all servers using a timer and act as follows}

CASE condition **of**

c-6.1. Timeout: **Write** "incomplete" record **to** the Log;

c-6.2. All servers answered (ACK):

Write "complete" record **to** the Log;

end;

c-7. C-COMMIT;

{End of Phase 2}

Server's Commit

Phase 1

s-1. **Await** a C-PREPARE message **from** the client **using** timer;

{The server will only pass this point if it has indeed received a C-PREPARE message from the client or timed out}

s-2. **If** not timed out **then** Vote ready or refuse;

s-3. **Case** condition **of**

```
        s-3.1.  ready:
                    begin
                        Write "ready" record to the Log;
                        Send a C-READY message to the client;
                        continue with Phase 2
                    end;
        s-3.2.  refuse:
                    begin
                        Write "refuse" record to the Log;
                        Send a C-REFUSE message to the client
                    end
        s-3.3.  Timeout:
                        Write "refuse"record to the Log
        end;
{End of Phase 1}
Phase 2
    s-4. Await a decision message (C-COMMIT or C-ROLLBACK) from the client
         using timer;
         {The server will only pass this point if it has indeed received a decision
         message from the client or times out}
    s-5. Case condition of
        s-5.1.  a C-COMMIT message was received:
                    begin
                        Write "commit" record to the Log;
                        Send ACK message to the client;
                        C-COMMIT
                    end;
        s-5.2.  a C-ROLLBACK message was received:
                    begin
                        Write "rollback" message to the Log;
                        C-ROLLBACK
                    end;
        s-5.3.  Timeout:
                    take whatever action to deal with blocking
        end;
{End of Phase 2}
```

Although it is not explicitly specified, each party can unilaterally decide for a C-ROLLBACK at any time if it has not (yet) voted "ready". We assume that the Client's Commit protocol is only initiated by the client after it has voted "ready". If a client never votes or votes "refuse", then it never sends a C-PREPARE message and the servers automatically time out eventually in their protocol. (We could have treated the client as a server in the protocol, but have chosen not to do so in order to save messages.) We require that C-COMMIT messages are ack'ed, but this can be omitted from the protocol without any harm. (We do not require that C-ROLLBACK messages are ack'ed, but could incorporate it easily if desired.) Voting essentially amounts to determining whether the local activities

in an atomic action have ended and properly been logged or not, but we only need it as an "abstract" operation. Where ever a "Log" is specified in the protocol, the local (client or server) log is meant. A "Phase 2" is not entered automatically after a "Phase 1", but only on an explicit "continue".

2.1 Correctness proof

It should be an interesting research topic to develop a completely formal "correctness proof" for the Individual Commit protocol. We outline a less formal proof here. We refer to line-numbers as c-0, c-1, etcetera.

Theorem A If no timeouts and no failures occur, then the Individual Commit protocol is correct.
Proof.
The requirements AC1 through AC5 are trivially satisfied. For AC2, note that if any party (client or server) decides spontaneously for C-ROLLBACK when it can, then all parties must follow suit eventually and cannot decide for anything else. AC6 is vacuously true. □

The next step is to consider the possibility of timeouts and a limited type of failures, namely "loss of protocol messages". In all cases except one, message loss necessarily leads to a timeout and thus it is sufficient to consider the latter only. The one exceptional case can arise in c-3, when some messages (including a C-REFUSE) have arrived but some have not and the clients acts on the C-REFUSE. In this case the client acts just like it would have in the case of timeout (line c-3.1.). Note that heavily delayed messages are considered "lost".

Theorem B If timeouts and loss of messages can occur but no server gets blocked, then the Individual Commit protocol is correct.
Proof.
The requirements AC1, AC3 and AC4 are trivially satisfied. For AC2, observe the following. If a server times out on s-1, it will never send a message and can only decide C-ROLLBACK (if it ever decides, cf. s-3.3.). The client necessarily executes c-3.1. and decides for C-ROLLBACK too. Other servers either time out on s-1 as well or receive the C-ROLLBACK decision in s-4. By assumption no server times out on s-4 (the blocked case). If the client times out on c-3.1., then it decides C-ROLLBACK and the servers can only reach the same decision by the very same argument. If the client times out on c-6.1., the only possible decision in the system is C-COMMIT and all servers must be in Phase 2. As we assume no blocking, all servers will eventually decide C-COMMIT. Thus AC2 is satisfied and, by the latter argument, AC5 as well. AC6 is vacuously true. □

In order to get any further we must some how deal with the problem of blocking. A blocked server timed out on s-5.3. and thus knows that it is blocked, but it is uncertain of the decision that may have been reached. In order to keep the Individual Commit protocol correct, a Server's Commit Termination Protocol must be added. The purpose of the Server's Commit Termination protocol is to enable a blocked server to determine the (apparent) decision reached in the system. Several possible strategies for a successful Server's Commit Termination protocol have been proposed, all based on polling. (For ex-

ample, if another server can be reached and it appears to have timed out on s-1, then the blocked server can decide C-ROLLBACK.) But no Server's Commit Termination protocol can guarantee that it will remove the possibility of blocking. (For example, if a blocked node is cut off permanently, it will forever remain uncertain.) We assume that any server can eventually reach the client again and thus a simple polling of the client will do as a Server's Commit Termination protocol (cf. requirement AC6). We conclude the following result.

Theorem C If timeouts and loss of messages can occur, then the Individual Commit protocol enhanced with the Server's Commit Termination protocol is correct.

The final step is to allow timeouts and arbitrary failures, i.e., "loss of protocol messages" and "site crashes". If a site crashes permanently, the Individual Commit protocol will simply continue in the remaining sites and perform as if the messages of the crashed site are lost from some point onwards. The Server's Commit Termination protocol should be extended in this case and somehow detect permanent crashes of the client. In general there is no guaranteed solution that avoids blocking, if permanent crashes can occur. Thus we assume that each site that crashes eventually recovers and resumes the commit protocol. We also assume that a site can actually recover to the point where it crashed, using the information in its log. The only problem now is to determine how to continue with the commit protocol, knowing that the other sites may have advanced in it after the crash. We present a possible recovery protocol below.

Commit Recovery Protocol

```
Client's Commit Recovery
cr-1. If the client crashed before c-4 then
          begin
              Write "rollback" record to the Log;
              Send C-ROLLBACK messages to all servers;
          end;
cr-2. If the client crashed after c-4 but before c-6.2. then
          begin
              Write "commit" record to the Log;
              Send C-COMMIT messages to all servers; activate timer;
              {Await answer messages from all servers using timer and act as follows }
              Case condition of
                 Timeout:                Write "incomplete" record to the Log;
                 All servers answered:   Write "complete" record to the Log
              end;
              C-COMMIT
          end;
cr-3. If the client crashed after c-6.2. then
          begin
              perform c-7 if necessary
          end;
```

Server's Commit Recovery

sr-1. **If** the server crashed before s-4 and without having executed s-3.1.
then
begin
C-ROLLBACK
end;

sr-2. **If** the server crashed after s-4 while being uncertain
then
begin
use the Server's Commit Termination protocol to remove blocking
end;

sr-3. **If** the server crashed after s-4 while being certain
then
begin
perform remaining activity if necessary in the C-COMMIT or
C-ROLLBACK
(whatever applies)
end;

We assume that **Write/Send** commands in the Individual Commit protocol are atomic, and thus no crash occurs "in between" a **Write** and the subsequent **Send**. (While the assumption is reasonable, it would nevertheless be of interest to analyse the protocol if this assumption is not made.)

Theorem D The Individual Commit Protocol enchanced with the Server's Commit Termination protocol and the Commit Recovery Protocol is a correct atomic commit protocol.
Proof.
If the client crashes before c-4, then each server either has C-ROLLBACK as the only option is uncertain. Thus cr-1 is a correct recovery action, in the sense of satisfying the requirements. If the client crashes in its Phase 2, then each server has either decided C-COMMIT or is uncertain. Replaying Phase 2 (in so far as it its Phase 1 and without sending a C-READY message, then the remaining sites will have progressed on the assumption that its messages are C-REFUSE or "lost". It means that the remaining sites are on their way to decide C-ROLLBACK, and sr-1 is fully consistent with this. If a server crashed after having sent a C-READY message, then either it had decided (and the decision is recovered) or is uncertain. Thus sr-2 and sr-3 are the actions to take. With the earlier analyses it easily follows that the complete protocol satisfies AC1 through AC6 and hence is correct. □

Utrecht, March 1989.

References

[1] Bernstein, P.A., V. Hadzilacos and N. Goodman, *Concurrency control and recovery in database systems* , Addison-Wesley Publ. Comp., Reading, Mass., 1987.

[2] Ceri, S., and G. Pelagatti, *Distributed databases - principles and systems* , McGraw-Hill Book Comp., New York, NY, 1984.

[3] Gentleman, W.M., *Message passing between sequential processes: the reply primitive and the administrator concept* , Software - P & E 11(1981) 435-466.

[4] Gray, J., *Notes on data base operating systems* , Report RJ2188, IBM Research Lab., San Jose, Ca., 1978.

[5] ISO, *Specification of protocols for application service elements - commitment, concurrency and recovery* , draft international standard, ISO TC97/DIS9805.2, 1985.

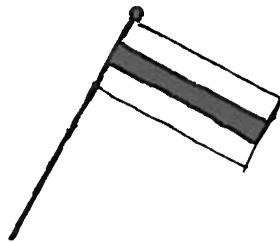
[6] Lampson, B., and H. Sturgis, *Crash recovery in a distributed data storage system* , Techn. Rep., Computer Science lab., Xerox - PARC, Palo Alto, Ca., 1976.

[7] van Leeuwen, J., *The Client/Server model in distributed computing*, Techn. Rep. RUU-CS-88-9, Dept. of Computer Science, University of Utrecht, Utrecht, 1988.

[8] Watson, R.W., *Distributed system architecture model, in: B.W. Lampson et al., Distributed systems - architecture and implementation* , Lect. Notes in Comput. Sci., vol. 105, Springer Verlag, Berlin, 1981, pp.10-43.

Voor Jaco

Graag Jaco, wil ik je feliciteren
met wat je hebt kunnen presteren
Om nu maar even iets te noemen
wil ik je als kok gaan roemen
Je Mexicaanse schotel is een succès fou
we tasten allen gaarne toe
Ook in het voorlezen ben je een kei
daarmee maak je de kleintjes steeds blij
Met de auto er op uit in de vacantie
is een ware aanslag op je tolerantie
Gelukkig weet je je in te tomen
zo bereik je het land van je dromen
Basta, het ogenblik is er nu
om over te stappen naar de VU
de l'université libre professeur:
een mooie titel, geen gezeur!
Last but not least: het CWI
vijfentwintig jaar! Hoera voor drie!



Carrien Lenstra

CORRESPONDENTIES IN SEMANTIEK

J.-J.Ch. Meyer
Vrije Universiteit, Amsterdam /
Katholieke Universiteit, Nijmegen

0. Samenvatting

In deze korte bijdrage wil ik trachten het hedendaagse werk aan semantiek van programmeertalen in een historisch/filosofisch perspectief te plaatsen, zodanig dat het duidelijk wordt dat de 'queeste' naar compositionele (denotationele) semantieken van moderne talen als POOL in feite direct voortvloeit uit de ideeën van Tarski (en in zekere zin al Frege) met betrekking tot de logica.

1. Semantiek

Semantiek (< σημαινω (Gr.) = met een teken aanduiden, betekenen) is *betekenisleer*: het houdt zich bezig met het betekenis geven aan syntactische objecten zoals woorden, zinnen, (logische) formules, en de laatste 20 jaren ook *programma's*. Ofschoon (praktisch) informatici vaak de indruk wekken geen behoefte te hebben aan een aparte discipline die de semantiek van programmeertalen bestudeert ("Ik weet gewoon wat een programma doet, en anders laat ik het wel uitvoeren om te zien wat het effect is"), is het wel degelijk zeer natuurlijk om een semantische theorie te ontwikkelen als een formeel model, waarin een programma geschreven in een bepaalde programmeertaal kan worden geïnterpreteerd.

Eigenlijk is dit niet anders dan in de linguistiek: de betekenis van een zin zou ook als "vanzelfsprekend" kunnen worden beschouwd. Linguisten hebben zich echter de moeite getroost om semantische theorieën voor natuurlijke talen op te stellen, teneinde grip te krijgen op de betekenis van de overweldigende hoeveelheid zinnen in de diverse talen van de wereld. We noemen hier de Franse taalkundige M. Bréal die de term "diachronische semantiek" in de taalkunde invoerde voor het probleemgebied van de oorsprong en verandering van betekenissen der woorden (Bréal [1964], Kuypers [1977]) en Richard Montague die een (compositionele) semantiek heeft ontworpen voor natuurlijke taal, gebaseerd op intensionele logica (cf. D.R. Dowty [1979]).

In feite is de behoefte om zich met betekenissen van syntactische objecten bezig te houden al veel ouder.

Van oudsher houdt de filosofie zich bezig met het waarheidsbegrip. Grofweg zijn er twee soorten waarheidsdefinities: gebaseerd op de *coherentietheorie* en op de *correspondentietheorie* (zie Haack [1978], Martin [1987]). De eerste theorie zegt dat een verzameling zinnen waar is als deze op de juiste manier *samenhangt*. Dit is natuurlijk nogal vaag. Van een samenhangende verzameling zinnen wordt op zijn minst vereist dat deze intern consistent is, maar verder ook dat deze als het ware een totaal wereldbeeld geeft.

Correspondentietheorie, anderszijds, beweert dat een zin waar is als deze op de een of andere

manier *correspondeert* met (entiteiten in) de wereld.Volgens een middeleeuwse formulering is een zin waar als wat deze betekent (dat wil zeggen, hetgeen waarmee deze correspondeert in de wereld), inderdaad zo is, en ook meer recent in Wittgenstein's *Tractatus* zien we een dergelijke correspondentie (cf. Wittgenstein [1922], Grayling [1988]). Uiteraard is dit in zijn algemeenheid ook erg vaag. De verdienste van *Tarski* is, dat deze correspondenties in een wiskundig kader heeft gegoten, zodat men precies kan vaststellen wat de correspondentie inhoudt.

Dat syntactische expressies corresponderen met entiteiten in de wereld, krijgt bij Tarski de gedaante van een wiskundige functie $[\cdot]$ van deze syntactische expressies $\mathcal{E}$ naar een semantisch *domein* $\mathcal{D}$ *van denotaties*. Dit domein $\mathcal{D}$ van denotaties (betekenissen) is een geformaliseerd model van de wereld (voor zover relevant met betrekking tot de beschouwde expressies). Bijvoorbeeld, arithmetische expressies (inclusief cijfers) worden afgebeeld op een domein van (mathematische) getallen; propositie-logische expressies worden afgebeeld op het domein van de (doorgaans twee) waarheidswaarden.

2. Compositionaliteit

Een belangrijk punt bij Tarski is de wijze waarop de semantische functie $[\cdot]$ wordt geëvalueerd. Laat e bijvoorbeeld een propositioneel-logische expressie zijn, dus $[e] \in \mathcal{D} = \{t, f\}$. Wil er sprake zijn van een echte correspondentietheorie van waarheid, dan moet er volgens Tarski een zin e^* in de metataal zijn die precies aangeeft wanneer e waar is, dat wil zeggen, wanneer $[e] = t$:

$$[e] = t \text{ desda } e^*.$$

e^* wordt de *waarheidsconditie* van e genoemd, die conditie die nodig en voldoende is voor de waarheid van expressie e. De waarheidsconditie e^* voor expressie e moet volgens Tarski ook voldoen aan het (ook al bij Frege voorkomende) *principe van compositionaliteit*: de denotatie van expressie e wordt volledig bepaald door de denotatie (cq. waarheid) van de subexpressies van e. Als bijvoorbeeld $e = e_1 \wedge e_2$, dan moet gelden dat

$$e^* \text{ d.e.s.d.a. } e_1^* \text{ en } e_2^*.$$

In het algemeen kunnen we dit principe van compositionaliteit als volgt formuleren:
Zij $e = e_1 \text{ op } e_2 \in \mathcal{E}$ voor een of ander syntactische connectief **op**, dan moet gelden voor $[\cdot]: \mathcal{E} \rightarrow \mathcal{D}$:

$$[e] = op\ ([e_1], [e_2]),$$

waarbij *op* een operator (functie) is van type $\mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ die correspondeert (met de werking / betekenis van) het connectief **op**. In het geval van $e = e_1 \wedge e_2$, leidt dit inderdaad tot

$e^* \Leftrightarrow [\![e]\!] = t \Leftrightarrow$

$\text{AND}([\![e_1]\!], [\![e_2]\!]) = t \Leftrightarrow$

$[\![e_1]\!] = t \text{ en } [\![e_2]\!] = t \Leftrightarrow e_1^* \text{ en } e_2^*$,

waarbij AND de boolean functie is, die correspondeert met het connectief $\wedge$, gedefiniëerd zoals gebruikelijk (bijvoorbeeld met een waarheidstabel).

3. Semantiek van programmeertalen

In de semantiek van programmeertalen, als onderzoeksterrein geïnitieerd in de eind 60-er jaren door pioniers zoals J.W. de Bakker, D. Scott en C. Strachey, zien we Tarski's principes weer terug in de context van programma's (zie De Bakker & Scott [1969][1], Scott & Strachey [1971], De Bakker [1980]).

Nu correspondeert onze verzameling $\mathcal{E}$ van expressies met een verzameling programma's in een bepaalde programmeertaal. (Aanvankelijk ALGOL-achtig, maar later werden ook andere talen beschouwd zoals logische programmeertalen (PROLOG), functionele talen (LISP, Miranda) en object-georiënteerde talen (POOL).) Het semantisch domein is in deze gevallen veel ingewikkelder geworden, met name voor talen met de mogelijkheid om recursieve en parallelle programma's te schrijven (zie bijvoorbeeld America & De Bakker [1988], De Bakker & Meyer [1988]). De grondgedachte is echter dezelfde: interpreteer expressies cq. programmastatements uit een klasse $\mathcal{E}$ in een (wiskundig) domein met behulp van een semantische functie $[\![\cdot]\!]: \mathcal{E} \to \mathcal{D}$, die (bij voorkeur) ***compositioneel*** is. $\mathcal{D}$ bestaat over het algemeen uit functies die de uitvoering van statements denoteren, bijvoorbeeld functies van inputwaarden van variabelen naar outputwaarden. Bijvoorbeeld, zij $e = e_1 ; e_2$, waarbij ; staat voor de sequentiële compositie (het achter elkaar uitvoeren), dan is

$$[\![e]\!] = [\![e_1]\!] \circ [\![e_2]\!],$$

waarbij de (mathematische) functie $\circ: \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ ruwweg werkt als volgt: $d_1 \circ d_2$ is het resultaat dat men krijgt door eerst d_1 toe te passen en dan d_2 op het resultaat hiervan.

Deze wijze van semantiek geven aan programma's heet *denotationele semantiek*: Ze werkt met behulp van denoterende semantische functies en is aldus een directe *specialisatie* van Tarskiaanse semantiek gebaseerd op *correspondentietheorie* voor programmeertalen, zoals Montague-semantiek dit is voor natuurlijke talen.

Met een beetje goede wil kunnen we in de context van programmeertalen ook de *coherentietheorie* terugvinden. Behalve denotationele semantiek is er ook nog zoiets als *operationele semantiek* van programma's. Deze geeft weer, hoe een programma zich operationeel (stapje voor stapje) gedraagt. Zo vaag gesteld, is het verschil met denotationele semantiek niet altijd even duidelijk, hetgeen vaak aanleiding geeft tot oeverloze discussies of een bepaalde semantiek nu operationeel dan wel denotationeel is. Laat ik omwille van de duidelijkheid hier stellen dat een denotationele semantiek wordt gedefiniëerd aan de hand van een semantische functie $[\![\cdot]\!]: \mathcal{E} \to \mathcal{D}$, die compositioneel is. Een operationeel semantiek wordt tegenwoordig vaak gedefiniëerd met

behulp van zogenaamde transitiesystemen en is dus *a priori* niet-compositioneel. (Dit neemt niet weg, dat er bij een operationeel gedefiniëerde semantiek vaak een semantische functie is te definiëren die de interpretatie (of liever evaluatie) van een statement m.b.v. het gegeven transitiesysteem weergeeft. Deze functie f kan zelfs *a posteriori* compositioneel zijn in de zin, dat men kan *bewijzen* dat f (e) = g(f(e_1), f(e_2)) voor e = e_1 **op** e_2 en een functie g: $\mathcal{D} \rightarrow \mathcal{D}$. Echter, een operationele semantiek wordt zo NIET *gedefiniëerd*!)

Op zichzelf staand kan men een operationele semantiek opvatten als een *coherentietheorie*: ze moet een samenhangend geheel zijn, intern consistent en een totale afspiegeling van de betreffende (programma)wereld.

Echter, hiermee is het verhaal niet uit. Juist omdat een operationele semantiek als primitiever wordt ervaren dan een denotationele semantiek–ze is/levert immers een directe afspiegeling van de werking van programma's–worden denotationele semantieken vaak gerechtvaardigd door een relatie vast te stellen met een operationele semantiek. In eenvoudige gevallen (sequentiële programmeertalen) is deze rechtvaardigende relatie vaak *(extensionele) equivalentie*: het opleveren van hetzelfde resultaat (dat wil zeggen hoewel '*intensioneel*' verschillend opgezet, hebben denotationele en operationele semantiek dezelfde *extensie*; cf. De Bakker [1980]). In moeilijker gevallen heeft men vaak een minder dwingende eis: de denotationele semantiek moet *volledig abstract* zijn ten opzichte van de operationele (zie bijv. Rutten [1988]). Dit komt dan neer op de eis dat de denotationele semantiek zoveel ingewikkelder uitkomsten mag geven dan de operationele, dat de denotationele semantiek compositioneel is en precies die statements identificeert die operationeel in elke context hetzelfde opleveren. Zodoende ontstaat er weer een *correspondentie* tussen (de twee) semantieken.

Noot

1. In feite ontstond dit werk uit een correspondentie tussen De Bakker en Scott met als doel het vinden van een complete calculus voor equivalenties tussen while-statements (Bron: persoonlijke 'correspondentie' (cq. gesprekken) met De Bakker, 1989). Dit werk leverde tevens de kern van een wiskundige theorie van programmacorrectheid, later verder ontwikkeld door De Bakker en De Roever (De Roever [1974], De Bakker [1980]), die een verbindend element zou vormen tussen de semantiek van programmeertalen in strikte zin en logica's voor programmacorrectheid, die in die tijd door onder meer Hoare werden voorgesteld onder de enigszins logisch contradictoire term "*axiomatische semantiek*" (zie Hoare [1969]).

Literatuur

1. P. America & J.W. de Bakker, Designing Equivalent Semantic Models for Process Creation, TCS 60, 1988, pp. 109 - 176.

2. J.W. de Bakker, Mathematical Theory of Program Correctness, Prentice-Hall Int., Englewood Cliffs, New Jersey, 1980.

3. J.W. de Bakker & J.-J.Ch. Meyer, Metric Semantics for Concurrency, BIT 28, 1988, pp. 504 - 529.

4. J.W. de Bakker & D. Scott, A Theory of Programs, Unpublished Notes, IBM Seminar, Vienna, 1969.

5. M. Bréal, Semantics: Studies in the Science of Meaning, 1964.

6. D.R. Dowty, Word Meaning and Montague Grammar, Reidel, Dordrecht, 1979.

7. A.C. Grayling, Wittgenstein, Oxford University Press, Oxford, 1988.

8. S. Haack, Philosophy of Logics, Cambridge University Press, Cambridge, 1978.

9. C.A.R. Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM 12, 1969, pp. 576-580.

10. K. Kuypers e.a., Encyclopedie van de filosofie, Winkler Prins Bibliotheek, Elsevier, Amsterdam/ Brussel, 1977.

11. J.N. Martin, Elements of Formal Semantics, Academie Press, Orlando, 1987.

12. W.P. de Roever, Recursion and Parameter Mechanisms: An Axiomatic Approach, in: Proceedings 2nd ICALP (J. Loeckx, ed.) LNCS 14, Springer, 1974, pp. 34-65.

13. J.J.M.M. Rutten, Correctness and Full Abstraction of Metric Semantics of Concurrency, CWI Rapport CS - R8831, 1988.

14. D.S. Scott & C. Strachey, Toward a Mathematical Semantics for Computer Languages, in: Proceedings Symposium on Computers and Automata (J. Fox, ed.), Polytechnic Institute of Brooklyn Press, 1971, pp. 19-46.

15. L. Wittgenstein, Tractatus Logico-philosophicus, F.R.S., London, 1922.

Università di Pisa DIPARTIMENTO DI INFORMATICA

Professor Jaco W. de Bakker
Stichting Mathematisch Centrum
Centrum voor Wiskunde en Informatica
Postbus 4079
1009 AB Amsterdam, The Netherlands

Pisa, January 28 1989

Dear Jaco:

I have heard you are celebrating your 25 years at CWI. I would like to send you my congratulations for your long activity in the field of semantics. Since the time you came in Pisa, more than twenty years ago, for a Conference organized by Alfonso Caracciolo di Forino, your work has been of extraordinary importance for all of us. I am sure you will continue to have a central role in making and organizing research in our field.

Best wishes!

Yours Sincerely

Ugo Montanari

ADDR : CORSO ITALIA, 40 - 56100 PISA - ITALY TEL.: +39 - (0)50 - 510111 TLX: 590291 DIPISA I E.MAIL: dipisa UUCP PART. IVA 00286820501

Boven de wolken moet de vrijheid wel grenzeloos zijn : alle angsten, alle zorgen en pijn blijven daaronder verborgen en zijn, met wat hier zo groot en machtig toeschijnt, ginder nietig en klein....

R. mey

Van harte gefeliciteerd met je 25 jarige C.W.I. jubileum!

Angeline

Views on Parallel Parsing: A preliminary survey

Anton Nijholt†

Faculteit Informatica, Universiteit Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands

ABSTRACT

A preliminary survey is presented of approaches to the parsing problem in parallel environments. The discussion includes parsing schemes which use more than one traditional parser, schemes where 'non-deterministic' choices during parsing are assigned separate processes, schemes where the number of processes depends on the length of the sentence being parsed, and schemes where the number of processes depends on the grammar size rather than on the input length.

1. Introduction

In the early 1970's papers appeared in which ideas on parallel compiling and executing of programs were investigated. In these papers parallel lexical analysis, syntactic analysis (parsing) and code generation were discussed. At that time various multiprocessor computers were introduced (CDC6500, 7600, STAR, ILLIAC IV, etc.) and the first attempts were made to construct compilers which used more than one processor when compiling programs (Lincoln[1970], Ellis[1971], and Zosel[1973]). In 1975 this activity led to a conference on *Programming Languages and Compilers for Parallel and Vector Machines* (see Donegan and Katz[1975] for parsing techniques for vector machines). Slowly, with the coming of new parallel architectures and the advent of VLSI, interest increased and presently interest in parallel compiling and executing is widespread. Although more slowly, a similar change of orientation occurred in the field of natural language processing. However, unlike the compiler construction environment with its generally accepted theories, in natural language processing no generally advocated theory of natural language analysis and understanding is available. Therefore it is not only the desire to exploit parallelism for the improvement of speed but it is also the assumption that human sentence processing is of an inherently parallel nature which makes computer linguists and

† With help from Bart Van Acker and Bart De Wolf, Faculteit Wetenschappen, Vrije Universiteit Brussel, Belgium.

cognitive scientists turn to parallel approaches for their problems. Hence, also in natural language processing environments there is presently a widespread interest in parallel processing techniques for natural language analysis.

Parallel parsing methods have been introduced in the areas of theoretical computer science, compiler construction and natural language processing. In the area of compiler construction these methods sometimes refer to the properties of programming languages, e.g. the existence of special keywords, the frequent occurrence of arithmetic expressions, etc. Sometimes the parsing methods that were introduced were closely related to existing and well known serial parsing methods, such as LL-, LR-, and precedence parsing. Parallel parsing was often looked upon as deterministic parsing of sentences with more than one serial parser. However, with the massively parallel architectures that have been designed and constructed, and with the possibility to design special-purpose chips for parsing and compiling, also the well known methods for general context-free parsing have again been investigated in order to see whether they allow parallel implementations. Typical results in this area are $O(n)$-time parallel parsing methods based on the Earley or the Cocke-Younger-Kasami parsing methods.

In order to study complexity results for parallel recognition and parsing of context-free languages theoretical computer scientists have introduced parallel machine models and special subclasses of the context-free languages (bracket languages, input-driven languages). Methods that have been introduced in this area aim at obtaining lower bounds for time and/or space complexity and are not necessarily useful from a more practical point of view. A typical result in this area tells us that context-free language recognition can be done in $O(log^2 n)$ time using n^6 processors, where n is the length of the input string.

In the area of natural language processing many kinds of approaches and results can be distinguished. While some researchers aim at cognitive simulation, others are satisfied with high performance language systems. The first mentioned researchers may ultimately ask for numbers of processors and connections between processors that approximate the number of neurons and connections between them in the human brain (that is, an order of 10^{11} neurons and 10^3–10^4 connections each). They model human language processing with connectionist models and therefore they are interested in massive parallelism and low degradation in the face of local errors. In connectionist approaches to parsing and natural language analysis the traditional methods of language analysis are often replaced by strongly interactive distributed processing of word senses, case roles and semantic markers (see e.g., Cottrell and Small[1984], Waltz and Pollack[1985] and Small[1987]). The second group of language researchers, those interested in designing and building high performance language systems, are interested in parallelism as well. For any system which has to understand natural language sentences it is necessary to distinguish different levels of analysis (see e.g. Nijholt[1988], where we distinguish the morphological, the lexical, the syntactic, the semantic, the referential and the behavioral level) and at each level a different kind of knowledge has to be invoked. Therefore we can distinguish different tasks: the application of morphological knowledge, the

application of lexical knowledge, etc. It is not necessarily the case that the application of one type of knowledge is under control of the application of an other type of knowledge. These tasks may interact and at times they can be performed simultaneously. Therefore processors which can work in parallel and which can communicate with each other can be assigned to these tasks.

In this paper various approaches to the problem of parallel parsing will be surveyed. We will discuss examples of parsing schemes which use more than one traditional parser, schemes where 'non-deterministic' choices during parsing are assigned separate processes, schemes where the number of processes depends on the length of the sentence being parsed, and schemes where the number of processes depends on the grammar size rather than on the input length.

2. From One to Many Traditional Serial Parsers

Introduction

As mentioned in the introduction, many algorithms for parallel parsing have been proposed. Concentrating on the ideas that underlie these methods, some of them will be discussed here. For an annotated bibliography containing references to other methods see Nijholt[1989]. Since we will frequently refer to *LR-parsing* a few words will be spent on this algorithm. The class of LR-grammars is a subclass of the class of context-free grammars. Each LR-grammar generates a *deterministic* context-free languages and each deterministic context-free language can be generated by an LR-grammar. From an LR-grammar an LR-parser can be constructed. The LR-parser consists of an LR-table and an LR-routine which consults the table to decide the actions that have to be performed on a pushdown stack and on the input. The pushdown stack will contain symbols denoting the *state* of the parser. As an example, consider the following context-free grammar:

1. S → NP VP
2. S → S NP
3. NP → *det *n
4. PP → *prep NP
5. VP → *v NP

With the LR-construction method the LR-table of Fig. 1 will be obtained from this grammar. It is assumed that each input string to be parsed will have an endmarker which consists of the $-sign.

An entry in the table of the form 'sh *n*' indicates the action 'shift state *n* on the stack and advance the input pointer'; entry 'r *n*' indicates the action 'reduce the stack using rule *n*'. The entry 'acc' indicates that the input string is accepted. The right part of the table is used to decide the state the parser has to go after a reduce action. In a reduce action states are popped from the stack. The number of states that are popped is equal to the length of the right hand side of the rule that has to be used in the reduction. With the state which becomes the topmost symbol of the stack (0–10)

state	*det	*n	*v	*prep	$	NP	PP	VP	S
0	sh3					2			1
1				sh5	acc		4		
2			sh6					7	
3		sh8							
4				re2	re2				
5	sh3					9			
6	sh3					10			
7				re1	re1				
8			re3	re3	re3				
9				re4	re4				
10				re5	re5				

Fig. 1 LR-parsing table for the example grammar.

and with the nonterminal of the left hand side of the rule which is used in the reduction (*S*, *NP*, *VP*, or *PP*) the right part of the table tells the parser what state to push next on the stack. In Fig. 2 the usual configuration of an LR-parser is shown.

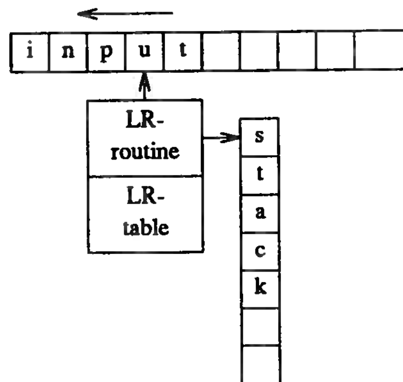


Fig. 2 LR-parser.

More than One Serial Parser

Having more than one processor, why not use two parsers? One of them can be used to process the input from left to right, the other can be used to process the input from right to left. Each parser can be assigned part of the input. When the parsers meet the complete parse tree has to be constructed from the partial parse trees delivered by the two parsers. Obviously, this idea is not new. We can find it in Tseytlin and Yushchenko[1977] and it appears again in Loka[1984]. Let $G=(N,\Sigma,P,S)$ be a context-free grammar. For any string $\alpha \in V^*$ let α^R denote the reversal of α. Let $G^R=(N,\Sigma,P^R,S)$ be the context-free grammar which is obtained from G by defining $P^R=\{i.\, A \rightarrow \alpha^R \mid i.\, A \rightarrow \alpha \in P\}$. It is not difficult to see that when we start a left-to-right top-down construction of a parse tree with respect to G at the leftmost symbol of a string w and a bottom-up right-to-left construction of a parse tree with respect to G^R at the rightmost symbol of w then, assuming the grammar is unambiguous, the

resulting partial parse trees can be tied together and a parse tree of w with respect to G is obtained. If the grammar is ambiguous all partial trees have to be produced before the correct combinations can be made. Similarly, we can start with a bottom-up parser at the left end of the string and combine it with a top-down parser starting from the right end of the string. Especially when grammar G allows a combination of a deterministic top-down (or LL-) parser and a deterministic bottom-up (or LR-) parser this might be a useful idea. However, in general we can not expect that if G is an LL-grammar that G^R is an LR-grammar and conversely.

Rather than having one or two parsers operating at the far left or the far right of the input, we would like to see a number of parsers, where the number depends on the 'parallelism' the input string allows, working along the length of the input string. If there is a natural way to segment a string, then each segment can have its own parser. Examples of this strategy are the methods described in Lincoln[1970], Mickunas and Schell[1975], Fischer[1975], Carlisle and Friesen[1985] and Lozinskii and Nirenburg[1986]. Here we confine ourselves to an explanation of Fischer's method. Fischer introduces 'synchronous parsing machines' (SPM) that LR-parse part of the input string. Each of the SPM's is a serial LR-parser. In theory the starting point of each SPM may be any symbol in the input string. For practical applications one may think of starting at keywords denoting the start of a procedure, a block, or even a statement. One obvious problem that is encountered is, when we let a serial LR-parser start somewhere in the input string, in what state should it start? The solution is to let each SPM carry a set of states, guaranteed to include the correct one. In addition, for each of these states the SPM carries a pushdown stack on which the next actions are to be performed. An outline of the parsing algorithm follows.

For convenience we assume that the LR-parser is an LR(0) parser. No look-ahead is necessary to decide a shift or a reduce action. In the algorithm M denotes the LR-parsing table and for any state s, $R(s)$ denotes the set consisting of the rule which has to be used in making a reduction in state s. By definition, $R(s)=\{0\}$ if no reduction has to be made in state s.

(1) *Initialization.*
Start one SPM at the far left of the input string. This SPM has only one stack and it only contains s_0, the start state. Start a number of other SPM's. Suppose we want to start an SPM immediately to the left of some symbol a. In the LR-parse table M we can find which states have a non-empty entry for symbol a. For each of these states the SPM which is started has a stack containing this state only. Hence, the SPM is started with just those states that can validly scan the next symbol in the string.

(2) *Scan the next symbol.*
Let a be the symbol to be scanned. For each stack of the SPM, if state s is on top, then
(a) if $M(s,a)=s'$ then push s' on the stack;
(b) if $M(s,a)=\emptyset$ then delete this stack from the set of stacks this SPM carries.
In the latter case the stack has been shown to be invalid. While scanning the

next input symbols the number of stacks that an SPM carries will decrease.

(3) *Reduce?*
Let $Q=\{s_1, \cdots, s_n\}$ be the set of top states of the stacks of the SPM under consideration. Define

$$R(Q)=\bigcup_{s\in Q} R(s).$$

(a) if $R(Q)=\{0\}$, then go to step (2); in this case the top states of the stacks agree that no reduction is indicated;
(b) if $R(Q)=\{i\}$, $i\neq 0$, and $i=A\rightarrow\gamma_i$, then, if the stacks of the SPM are deep enough to pop off $|\gamma_i|$ states and not be empty, then do reduction i;
(c) otherwise, if we have insufficient stack depth or not all top states agree on the same reduction, we stop this SPM (for the time being) and, if possible, we start a new SPM to the immediate right.

An SPM which has been stopped can be restarted. If an SPM is about to scan a symbol already scanned by an SPM to its immediate right, then a merge of the two SPM's is attempted. The following two situations have to be distinguished:

- If the left SPM contains a single stack with top state s then s is the correct state to be in and we can select from the stacks of the right SPM the stack with bottom state s. Pop s from the left stack and then concatenate the two. All other stacks can be discarded and the newly obtained SPM can continue parsing.
- If the left SPM contains more than one stack then it is stopped. It has to wait until it is restarted by an SPM to its left. Notice that the leftmost SPM always has one stack and it will always have sufficient stack depth. Therefore there will always be an SPM coming from the left which can restart a waiting SPM.

In step (3c) we started a new SPM immediate to the right of the stopped SPM. What set of states and associated stacks should it be started in? We can not, as was done in the initialization, simply take those states which allow a scan of the next input symbol. To the left of this new SPM reductions may have been done (or will be done) and therefore other states should be considered in order to guarantee that the correct state is included. Hence, if in step (3) $|R(Q)|>1$ then for each s in Q, if $R(s)=\{0\}$, then add s to the set of states of the new SPM and if $R(s)=\{i\}$ add to the set of states that have to be carried by the new SPM also the states that can become topmost after a reduction using production rule i (perhaps followed by other reductions).

This concludes our explanation of Fischer's method. For more details and extensions of these ideas the reader is referred to Fischer[1975].

'Solving' Parsing Conflicts by Parallelism?

To allow more efficient parsing methods restrictions on the class of general context-free grammars have been introduced. These restrictions have led to, among others, the classes of LL-, LR- and precedence grammars and associated LL-, LR- and precedence parsing techniques. The LR-technique uses, as discussed in the previous

section, an LR-parsing table which is constructed from the LR-grammar.

If the grammar from which the table is constructed is not an LR-grammar, then the table will contain conflict entries. With a conflict entry the parser has to choose. One decision may turn out to be wrong or both (or more) possibilities may be correct but only one may be chosen. The entry may allow reduction of a production rule but at the same time it may allow shifting of the next input symbol onto the stack. A conflict entry may also allow reductions according to different production rules. Consider the following example grammar *G*:

1. S → NP VP
2. S → S NP
3. NP → *n
4. NP → *det *n
5. NP → NP PP
6. PP → *prep NP
7. VP → *v NP

The parsing table for this grammar, taken from Tomita[1985], is shown in Fig. 3.

state	*det	*n	*v	*prep	$	NP	PP	VP	S
0	sh3	sh4				2			1
1				sh6	acc		5		
2			sh7	sh6			9	8	
3		sh10							
4			re3	re3	re3				
5				re2	re2				
6	sh3	sh4				11			
7	sh3	sh4				12			
8				re1	re1				
9			re5	re5	re5				
10			re4	re4	re4				
11			re6	re6,sh6	re6		9		
12				re7,sh6	re7		9		

Fig. 3 LR-parsing table for grammar G.

Tomita's answer to the problem of LR-parsing of general context-free grammars is 'pseudo-parallelism'. Each time during parsing the parser encounters a multiple entry, the parsing process is split into as many processes as there are entries. Splitting is done by replicating the stack as many times as necessary and then continue parsing with the actions of the entry. The processes are 'synchronized' on the shift action. Any process that encounters a shift action waits until the other processes also encounter a shift action. Therefore all processes look at the same input word of the sentence.

Obviously, this LR-directed breadth-first parsing may lead to a large number of non-interacting stacks. So it may occur that during parts of a sentence all processes behave in exactly the same way. Both the amount of computation and the amount of space can be reduced considerably by unifying processes by combining their stacks into a so-called 'graph-structured' stack. Tomita does not suggest a

parallel implementation of the algorithm. Rather his techniques for improving efficiency are aimed at efficient serial processing of sentences. Nevertheless, we can ask whether a parallel implementation might be useful. Obviously, Tomita's method is not a 'parallel-designed' algorithm. There is a master routine (the LR-parser) which maintains a data structure (the graph-structured stack) and each word that is read by the LR-parser is required for each process (or stack). In a parallel implementation nothing is gained when we weave a list of stacks into a graph-structured stack. In fact, when this is done, Tomita's method becomes closely related to Earley's method (see section 4) and it seems more natural – although the number of processes may become too high – to consider parallel versions of this algorithm since it is not in advance restricted by the use of a stack. When we want to stay close to Tomita's ideas, then we rather think of a more straightforward parallel implementation in which each conflict causes the creation of a new LR-parser which receives a copy of the stack and a copy of the remaining input (if it is already available) and then continues parsing without ever communicating with the other LR-parsers that work on the same string. On a transputer network, for example, each transputer may act as an LR-parser. However, due to its restrictions on interconnection patterns, sending stacks and strings through the network may become a time-consuming process. When a parser encounters a conflict the network should be searched for a free transputer and stack and remainder of the input should be passed through the network to this transputer. This will cause other processes to slow down and one may expect that only a limited 'degree of non-LR-ness' will allow an appropriate application of these ideas. Moreover, one may expect serious problems when on-line parsing of the input is required.

3. Translating Grammar Rules into Process Configurations

A simple 'object-oriented' parallel parsing method for ε-free and cycle-free context-free grammars has been introduced by Yonezawa and Ohsawa[1988]. The method resembles the well known Cocke-Younger-Kasami parsing method, but does not require that the grammars are in Chomsky Normal Form (CNF). Consider again our example grammar G:

1. S → NP VP
2. S → S NP
3. NP → *n
4. NP → *det *n
5. NP → NP PP
6. PP → *prep NP
7. VP → *v NP

This set of rules will be viewed as a network of concurrently working computing agents. Each occurrence of a (pre-)terminal or a nonterminal symbol in the grammar rules corresponds with an agent with modest processing power and internal memory. The agents communicate with one another by passing subtrees of possible parse trees. The topology of the network is obtained as follows. Rule 1 yields the network

fragment depicted in Fig. 4.

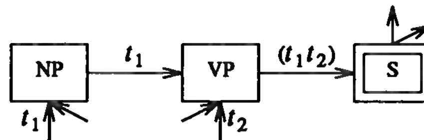


Fig. 4 From rules to configuration.

In the figure we have three agents, one for *NP*, one for *VP* and a 'double' agent for *S*. Suppose the *NP*-agent has received a subtree t_1. It passes t_1 to the *VP*-agent. Suppose this agent has received a subtree t_2. It checks whether they can be put together (the 'boundary adjacency test') and if this test succeeds it passes $(t_1\ t_2)$ to the *S*-agent. This agent constructs the parse tree $(S\ (t_1\ t_2))$ and distributes the result to all computing agents in the network which correspond with an occurrence of *S* in a right hand side of a rule. The complete network for the rules of *G* is shown in Fig. 5. As can be seen in the network, there is only one of these *S*-agents. For this agent $(S\ (t_1\ t_2))$ plays the same role as t_1 did for the *NP*-agent. If the boundary adjacency test is not successful, then the *VP*-agent stores the trees until it has a pair of trees which satisfies the test.

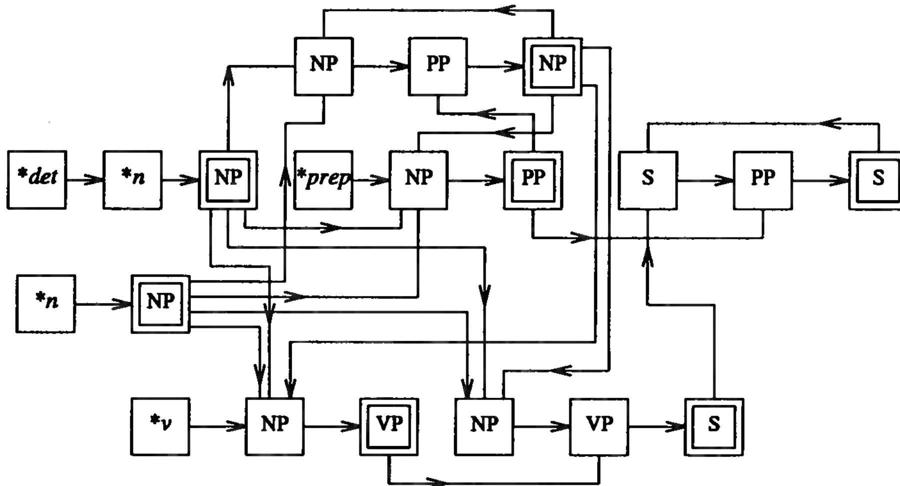


Fig. 5 Computing agents for grammar *G*.

As an example, consider the sentence *The man saw a girl with a telescope.* For this particular sentence we do not want to construct from a subtree t_1 for *a telescope* and from a subtree t_2 for *saw the girl* a subtree for *a telescope saw a girl,* although the rule $S \rightarrow NP\ VP$ permits this construction. Therefore, words to be sent into the network are provided with tags representing positional information and during construction of a subtree this information is inherited from its constituents. For our example sentence the input should look as

(0 1 the)(1 2 man)(2 3 saw)(3 4 a)(4 5 girl)(5 6 with)(6 7 a)(7 8 telescope).

Combination of tokens and trees according to the grammar rules and the positional information can yield a subtree *(3 5 (NP ((*det a)(*n girl))))* but not a subtree in which *(0 1 the)* and *(4 5 girl)* are combined. Each word accompanied with its tags is distributed to the agents for its (pre-)terminal(s) by a *manager agent* which has this information available.

If the context-free grammar which underlies the network is ambiguous then all possible parse trees for a given input sentence will be constructed. It is possible to pipe-line constructed subtrees to semantic processing agents which filter the trees so that only semantically valid subtrees are distributed to other agents. An other useful extension is the capability to unparse a sentence when the user of a system based on this method backspaces to previously typed words. This can be realized by letting the agents send anti-messages that cancel the effects of earlier messages. It should be noted that the parsing of a sentence does not have to be finished before a next sentence is fed into the network. By attaching another tag to the words it becomes possible to distinguish the subtrees from one sentence from those of an other sentence.

The method as explained here has been implemented in the object-oriented concurrent language ABCL/1. For the experiment a context-free English grammar which gave rise to 1124 computing agents has been used. Sentences with a length between 10 and 30 words and a parse tree height between 10 and 20 were used for input. Parallelism was simulated by time-slicing. From this simulation it was concluded that a parse tree is produced from the network in $O(n \times h)$ time, where n is the length of the input string and h is the height of the parse tree. Obviously, simple examples of grammars and their sentences can be given which cause an explosion in the number of adjacency tests and also in the number of subtrees that will be stored without ever being used. Constructs which lead to such explosions are normally not seen in context-free descriptions of natural language. As an example, consider the method described above for a context-free grammar with rules $S \rightarrow aS$ and $S \rightarrow a$ only.

There are several ways in which the number of computing agents can be reduced. For example, instead of the three double *NP*-agents of Fig. 5 it is possible to use one double *NP*-agent with the same function but with an increase of parse trees that have to be constructed and distributed. The same can be done for the two *S*-agents. A next step is to eliminate all double agents and give their tasks to the agents which correspond with the rightmost symbol of a grammar rule. It is also possible to have one computing agent for each grammar rule. In this way we obtain the configuration of Fig. 6. It will be clear what has to be done by the different agents.

Another configuration with a reduced number of computing agents is obtained if we have an agent for each nonterminal symbol of the grammar. For the example grammar we have four agents, the *S*-, the *NP*-, the *VP*-, and the *PP*-agent. If we want we can also introduce agents for the pre-terminals or even for each word which can occur in an input sentence. We confine ourselves to agents for the nonterminal

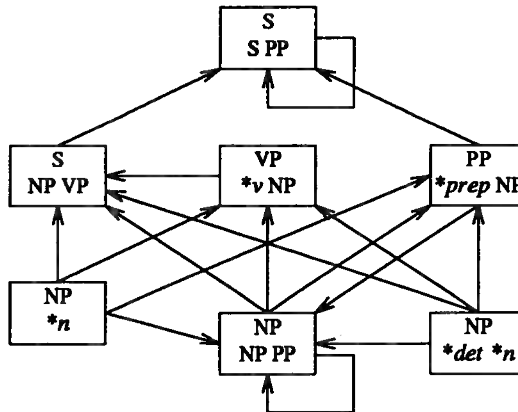


Fig. 6 Agents for grammar rules.

symbols and discuss their roles. In Fig. 7 we have displayed the configuration of computing agents which will be obtained from the example grammar.

The communication between the agents of this network is as follows.

(1) The *S*-agent sends subtrees with root *S* to itself; it receives subtrees from itself, the *PP*-agent, the *NP*-agent, and the *VP*-agent.

(2) The *NP*-agent sends subtrees with root *NP* to itself, the *S*-agent, the *VP*-agent and the *PP*-agent; it receives subtrees from itself and from the *PP*-agent; moreover, input comes from the manager agent.

(3) The *VP*-agent sends subtrees with root *VP* to the *S*-agent; it receives subtrees from the *NP*-agent; moreover, input comes from the manager agent.

(4) The *PP*-agent sends subtrees with root *PP* to the *S*-agent and to the *NP*-agent; it receives subtrees from the *NP*-agent; moreover, it receives input from the manager agent.

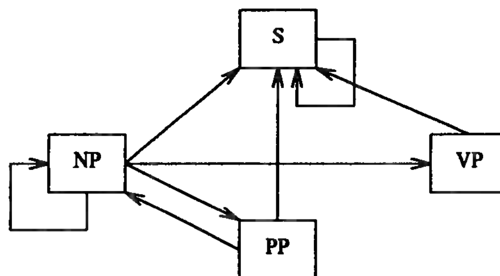


Fig. 7 Agents for nonterminal symbols.

The task of each of these nonterminal-agents is to check whether the subtrees it receives can be put together according to the grammar rules with the nonterminal as left hand side and according to positional information that is carried along with the

subtrees. If possible, a tree with the nonterminal as root is constructed, otherwise the agent checks other trees or waits until trees are available.

4. From Sentence Words to Processes

Cocke-Younger-Kasami's Algorithm

Traditional parsing methods for context-free grammars have been investigated in order to see whether they can be adapted to a parallel processing view. In Chu and Fu[1982] parallel aspects of the tabular Cocke-Younger-Kasami algorithm have been discussed. The input grammar should be in CNF, hence, each rule is of the form $A \rightarrow BC$ or $A \rightarrow a$. This normal form allows the following bottom-up parsing method. For any string $x = a_1 a_2 \cdots a_n$ to be parsed a strictly upper-triangular $(n+1)\times(n+1)$ recognition table T is constructed. Each table entry $t_{i,j}$ will contain a subset of N (the set of nonterminal symbols) such that $A \in t_{i,j}$ if and only if $A \Rightarrow^* a_{i+1} \cdots a_j$. String $x \in L(G)$ if and only if $S \in t_{0,n}$ when construction of the table is completed:

(1) Compute $t_{i,i+1}$, as i ranges from 0 to $n-1$, by placing A in $t_{i,i+1}$ exactly when there is a production $A \rightarrow a_{i+1}$ in P.

(2) Set d=1. Assuming $t_{i,i+d}$ has been formed for $0 \leq i \leq n-d$, increase d with 1 and compute $t_{i,j}$ for $0 \leq i \leq n-d$ and $j=i+d$ where A is placed in $t_{i,j}$ when, for any k such that $i+1 \leq k \leq j-1$, there is a production $A \rightarrow BC \in P$ with $B \in t_{i,k}$ and $C \in t_{k,i+j}$.

In this form the algorithm is usually presented (see e.g. Graham and Harrison [1976]). Fig. 8 may be helpful in understanding a parallel implementation.

	0,1	0,2	0,3	0,4	0,5
		1,2	1,3	1,4	1,5
			2,3	2,4	2,5
				3,4	3,5
					4,5

Fig. 8 Strictly upper-triangular CYK-table.

Notice that after step (1) the computation of the entries is done diagonal by diagonal until entry $t_{0,n}$ has been completed. For each entry of a diagonal only elements of preceding diagonals are used to compute its value. More specifically, in order to see whether a nonterminal should be included in an element $t_{i,j}$ it is necessary to compare $t_{i,k}$ and $t_{k,j}$, with k between i and j. The amount of storage that is required by

this method is proportional to n^2 and the number of elementary operations is proportional to n^3. Unlike Yonezawa and Oshawa's algorithm where positional information needs an explicit representation, here it is in fact available (due to the CNF of the grammar) in the indices of the table elements.

From the recognition table we can conclude a two-dimensional configuration of processes. For each entry $t_{i,j}$ of the strictly upper-triangular table there is a process $P_{i,j}$ which receives table elements (i.e., sets of nonterminals) from processes $P_{i,j-1}$ and $P_{i+1,j}$. Process $P_{i,j}$ transmits the table elements it receives from $P_{i,j-1}$ to $P_{i,j+1}$ and the elements it receives from $P_{i+1,j}$ to $P_{i-1,j}$. Process $P_{i,j}$ transmits the table element it has constructed to processes $P_{i-1,j}$ and $P_{i,j+1}$. Fig. 9 shows the process structure for $n=5$. As soon as a table element is computed or available it is sent to its right and upstairs neighbor. Each process should be provided with a coding of the production rules of the grammar. Clearly, each process requires $O(n)$ time. It is not difficult to see that like similar algorithms suitable for VLSI-implementation, e.g. systolic algorithms for multiplication or transitive closure computation (see Guibas et al[1979] and many others) the required parsing time is also $O(n)$. In Chu and Fu[1982] a VLSI design for this algorithm is presented (see also Tan[1983]).

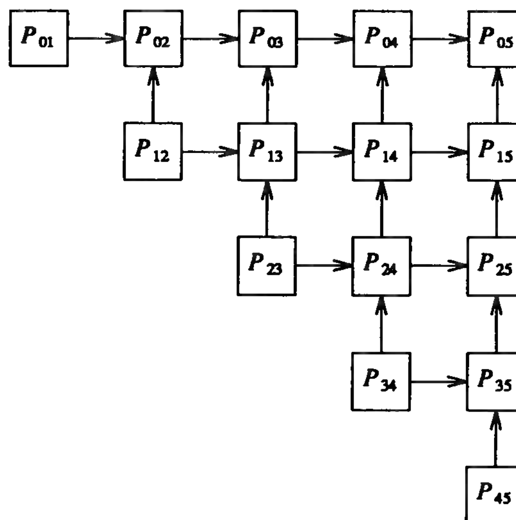


Fig. 9 Process configuration for CYK's algorithm.

Earley's Algorithm

The second algorithm we discuss in this section is the well known Earley's method. For general context-free grammars Earley parsing takes $O(n^3)$ time. This time can be reduced to $O(n^2)$ or $O(n)$ for special subclasses of the general context-free grammars. Many versions of Earley's method exist. In Graham and Harrison[1976] the following tabular version can be found. For any string $x = a_1 a_2 \cdots a_n$ to be parsed an upper-triangular $(n+1)\times(n+1)$ recognition table T is constructed. Each table entry

$t_{i,j}$ will contain a set of items, i.e., a set of elements of the form $A \rightarrow \alpha \cdot \beta$ (a dotted rule), where $A \rightarrow \alpha\beta$ is a production rule from the grammar and the dot $\cdot$ is a symbol not in $N \cup \Sigma$. The computation of the table entries goes column by column. The following two functions will be useful. Function PRED:$N \rightarrow 2^D$, where $D = \{A \rightarrow \alpha \cdot \beta \mid A \rightarrow \alpha\beta \in P\}$, is defined as

$$\text{PRED}(A) = \{B \rightarrow \alpha \cdot \beta \mid B \rightarrow \alpha\beta \in P,\ \alpha \Rightarrow^* \varepsilon \text{ and there exist } \gamma \in V^* \text{ with } A \Rightarrow^* B\gamma\}.$$

Function PREDICT:$2^N \rightarrow 2^D$ is defined as

$$\text{PREDICT}(X) = \bigcup_{A \in X} \text{PRED}(A).$$

Initially, $t_{0,0}$=PREDICT($\{S\}$) and all other table entries are empty. Suppose we want to compute the elements of column j, $j>0$. In order to compute $t_{i,j}$ with $i \neq j$ assume that all elements of the columns of the upper-triangular table to the left of column j have already been computed and in column j the elements $t_{k,j}$ for $j-1 \leq k \leq i+1$ have been computed.

(1) Add $B \rightarrow \alpha a \beta \cdot \gamma$ to $t_{i,j}$ if $B \rightarrow \alpha \cdot a \beta \gamma \in t_{i,j-1}$, $a = a_j$ and $\beta \Rightarrow^* \varepsilon$.

(2) Add $B \rightarrow \alpha a \beta \cdot \gamma$ to $t_{i,j}$, if, for any k such that $j-1 \leq k \leq i+1$, $B \rightarrow \alpha \cdot A \beta \gamma \in t_{i,k}$, $A \rightarrow \omega \cdot \in t_{k,j}$ and $\beta \Rightarrow^* \varepsilon$.

(3) Add $B \rightarrow \alpha A \beta \cdot \gamma$ to $t_{i,j}$ if $B \rightarrow \alpha \cdot A \beta \gamma \in t_{i,i}$, $\beta \Rightarrow^* \varepsilon$ and there exists $C \in N$ such that $A \Rightarrow^* C$ and $C \rightarrow \omega \cdot \in t_{i,j}$.

After all elements $t_{i,j}$ with $0 \leq i \leq j-1$ of column j have been computed then it is possible to compute $t_{j,j}$:

(4) Let $X_j = \{A \in N \mid B \rightarrow \alpha \cdot A \beta \in t_{i,j}, 0 \leq i \leq j-1\}$. Then $t_{j,j}$=PREDICT(X_j).

Fig. 10 illustrates the four steps by showing which elements provide the information for computing $t_{i,j}$ and $t_{j,j}$. It is not difficult to see that $A \rightarrow \alpha \cdot \beta \in t_{i,j}$ if and only if there exists $\gamma \in V^*$ such that $S \Rightarrow^* a_1 \cdots a_i A \gamma$ and $\alpha \Rightarrow^* a_{i+1} \cdots a_j$. Hence, in $t_{0,n}$ we can read whether the sentence was correct. The algorithm can be extended in order to produce parse trees.†

Various parallel implementations of Earley's algorithm have been suggested in the literature (see e.g. Chiang and Fu[1982], Tan[1983] and Sijstermans[1986]). The algorithms differ mainly in details on the handling of ε-rules, preprocessing, the representation of data and circuit and layout design. The main problem in a parallel implementation of the previous algorithm is the computation of the diagonal elements $t_{i,i}$, for $0 \leq i \leq n$. The solution is simple. Initially all elements $t_{i,i}$, $0 \leq i \leq n$, are set equal to PREDICT(N), where N is the set of nonterminal symbols. The other entries are defined according to the steps (1), (2) and (3). As a consequence, we now have $A \rightarrow \alpha \cdot \beta \in T_{i,j}$ if and only if $\alpha \Rightarrow^* a_{i+1} \cdots a_j$. In spite of weakening the conditions on the contents of the table entries the completed table can still be used to

† When Earley's algorithm was introduced it was compared with the exponential time methods in which successively every path was followed whenever a non-deterministic choice occurred. Since in Earley's algorithm a 'simultaneous' following of paths can be recognized, it was sometimes considered as a parallel implementation of the earlier depth-first algorithms (see e.g. Lang[1971]).

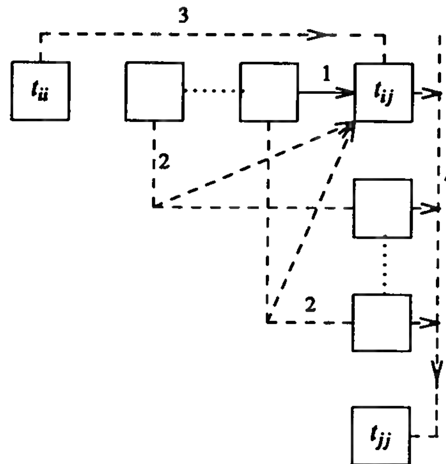


Fig. 10 Computation of t_{ij} and t_{jj} in Earley's algorithm.

determine whether an input sentence was correct. Moreover, computation of the elements can be done diagonal by diagonal, similar to the CYK-algorithm.

(1) Set $t_{i,i}$ equal to PREDICT(N), $0 \leq i \leq n$.

(2) Set d=0. Assuming $t_{i,i+d}$ has been formed for $0 \leq i \leq n-d$, increase d with 1 and compute $t_{i,j}$ for $0 \leq i \leq n-d$ and $j=i+d$ according to:

 (2.1) Add $B \to \alpha a \beta \cdot \gamma$ to $t_{i,j}$ if $B \to \alpha \cdot a \beta \gamma \in t_{i,j-1}$, $a=a_j$ and $\beta \Rightarrow^* \varepsilon$.

 (2.2) Add $B \to \alpha a \beta \cdot \gamma$ to $t_{i,j}$, if, for any k such that $j-1 \leq k \leq i+1$, $B \to \alpha \cdot A \beta \gamma \in t_{i,k}$, $A \to \omega \cdot \in t_{k,j}$ and $\beta \Rightarrow^* \varepsilon$.

 (2.3) Add $B \to \alpha A \beta \cdot \gamma$ to $t_{i,j}$ if $B \to \alpha \cdot A \beta \gamma \in t_{i,i}$, $\beta \Rightarrow^* \varepsilon$ and there exists $C \in N$ such that $A \Rightarrow^* C$ and $C \to \omega \cdot \in t_{i,j}$.

VLSI designs or process configurations which implement this algorithm in such a way that it takes $O(n)$ time (with $O(n^2)$ cells or processes can be found in Chiang and Fu[1982], Tan[1983] and Sijstermans[1986] (see also Fig. 9 and its explanation).

Natural Language Applications

In the field of natural language processing the Earley method is well known. Sometimes closely related methods such as (active) chart parsing (see Winograd[1983]) are used. Because of this close relationship a parallel implementation along the lines sketched above is possible. A similar remark holds for parsing augmented transition networks (ATN's). Earley's algorithm can be modified to transition networks and extended to ATN's (see e.g. Chou and Fu[1975]). It seems worthwhile to explore the possibilities of parallel Earley parsing for natural language applications.

Transformational Development of Parallel Parsing Algorithms

An interesting view on constructing correct parsing algorithms that are executable on parallel architectures is given by Partsch[1984b]. This view is part of a more general methodology of program development by transformations: develop a program from a formal specification by stepwisely applying correctness-preserving transformation rules (see e.g. Bauer and Wössner[1982]). In Partsch[1984a] this approach is applied to the development of the serial Earley algorithm (see also Jones[1971]) and in Partsch[1984b] the same approach is presented for the development of a parallel version of the Cocke-Younger-Kasami algorithm, and it is remarked that the same development strategy can be used to obtain a parallel version of Earley's algorithm. Without going into details we mention that with the help of some general transformation principles a tabular version of Cocke-Younger-Kasami's algorithm is obtained which is well suited for execution on a vector machine. With n processors the algorithm takes $O(n^2)$ time for context-free grammars in Chomsky Normal Form. If there are less than n processors a combination of parallel and sequential computation of the table elements can be done.

5. Conclusions

A survey of some ideas in parallel parsing has been presented. No attention has been paid to ideas aimed at improving upper bounds for the recognition and parsing of general context-free languages. An introduction to that area can be found in Chapter 4 of Gibbons and Rytter[1988]. Neither have we been looking here at the connectionist approaches in parsing and natural language processing. In an extended version of this paper context-free parsing in connectionist networks will be discussed. More references to papers on parallel parsing can be found in Nijholt et al[1989].

6. References

Bauer, F.L. and H. Wössner [1982]. *Algorithmic Language and Program Development*. Springer-Verlag, Berlin, 1982.

Carlisle, W.H. and D.K. Friesen [1985]. Parallel parsing using Ada. Proceedings 3rd Annual National *Conference on Ada Technology*, March 1985, 103-106.

Chiang, Y.T. and K.S. Fu [1982]. A VLSI architecture for fast context-free language recognition (Earley's algorithm). Proceedings Third International *Conf. on Distributed Comp. Systems*, 1982, 864-869.

Chou, S.M. and K.S. Fu [1975]. Transition networks for pattern recognition. School for Electrical Engineering, Purdue University, West Lafayette, Indiana, TR-EE 75-39, 1975.

Chu, K.-H. and K.S. Fu [1982]. VLSI architectures for high-speed recognition of context-free languages and finite-state languages. Proceedings of the Ninth Annual *Symposium on Computer Architectures*, *SIGARCH Newsletter* 10 (1982), No.3, 43-49.

Cottrell, G.W. and S.L. Small [1984]. Viewing parsing as word sense discrimination: A connectionist approach. In: *Computational Models of Natural Language Processing*, B.G. Bara and G. Guida (eds.), Elsevier Science Publishers, North-Holland, 1984, 91-119.

Donegan, M.K. and S.W. Katzke [1975]. Lexical analysis and parsing techniques for vector machines. In: Proceedings Conference on *Programming Languages and Compilers for Parallel and Vector Machines. SIGPLAN Notices* 10 (1975), No.3, 138-145.

Ellis, C.A. [1971]. Parallel compiling techniques. Proc. *Annual ACM Conf.* 1971, New York, 508-519.

Fischer, C.N. [1975]. Parsing context-free languages in parallel environments. Ph.D. Thesis, Tech. Report 75-237, Dept. of Computer Science, Cornell University, 1975.

Gibbons, A. and W. Rytter [1988]. Parallel recognition and parsing of context-free languages. Chapter 4 in *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, 1988.

Graham, S.L. and M.A. Harrison [1976]. Parsing of general context-free languages. *Advances in Computers*, Vol. 14, M. Yovits and M. Rubinoff (eds.), Academic Press, New York, 1976, 76-185.

Guibas, L.J., H.T. Kung and C.D. Thompson [1979]. Direct VLSI implementation of combinatorial algorithms. *Proc. Conf. on VLSI*, Caltech, January 1979, 509-526.

Jones, C.B. [1971]. Formal development of correct algorithms. An example based on Earley's recognizer. Proc. of ACM SIGPLAN Conf. on *Proving Assertions about Programs*, 1971, 150-169.

Kindervater, G.A.P. and J.K. Lenstra [1985]. An introduction to parallelism in combinatorial optimization. In: *Parallel Computers and Computations*, J.K. van Leeuwen and J.K. Lenstra (eds.), CWI-Syllabus 9, Centre for Mathematics and Computer Science, Amsterdam, 1985, 163-184.

Lang, B. [1971]. Parallel non-deterministic bottom-up parsing. In: Proc. *Int. Symposium on Extensible Languages*. Grenoble, 1971, *SIGPLAN Notices* 6, Nr. 12, December 1971.

Lincoln, N. [1970]. Parallel programming techniques for compilers. *SIGPLAN Notices* 5 (1970), No.10, 18-31.

Loka, R.R. [1984]. A note on parallel parsing. *SIGPLAN Notices* 19 (January 1984), 57-59.

Lozinskii, E.L. and S. Nirenburg [1986]. Parsing in parallel. *Computer Languages* 11 (1986), 39-51.

Mickunas, M.D. and R.M. Schell [1978]. Parallel compilation in a multiprocessor environment. Proceedings *ACM Annual Conf.*, 1978, 241-246.

Nijholt, A. [1988]. *Computers and Languages: Theory and Practice.* Studies in Computer Science and Artificial Intelligence. North-Holland, Elsevier Science Publishers, Amsterdam, 1988.

Nijholt, A. et al [1989]. An annotated bibliography on parallel parsing. Twente University, Internal Memorandum, in preparation, 1989.

Partsch, H. [1984a]. Structuring transformational developments: a case study based on Earley's recognizer. *Sci. Comput. Program.* 4 (1984), 17-44.

Partsch, H. [1984]. Transformational derivation of parsing algorithms executable on parallel architectures. In: *Programmiersprachen und Programmentwicklung*, 8. Fachtagung, Zürich, März 1984, Informatik Fachberichte 77, U. Ammann (ed.), Springer-Verlag, Berlin, 1984.

Rytter, W. [1987]. On the complexity of parallel parsing of general context-free languages. *Theoretical Computer Science* 47 (1987), 315-322.

Schell Jr., R.M. [1979]. Methods for constructing parallel compilers for use in a multi-processor environment. Ph.D. Thesis, Dept. of Computer Science, Report No. 958, University of Illinois at Urbana-Champaign, February 1979.

Small, S.L. [1987]. A distributed word-based approach to parsing. In: *Natural Language Parsing Systems.* L. Bolc (ed.), Springer-Verlag, Berlin, 1987, 161-201.

Srikant, Y.N. and P. Shankar [1987]. Parallel parsing of programming languages. *Information Sciences* 43 (1987), 55-83.

Sijstermans, F.W. [1986]. Parallel parsing of context-free languages. Doc. No. 202, Esprit Project 415, Subproject A: Object-oriented language approach, Philips Research Laboratories, Eindhoven, 1986.

Tan, H.D.A. [1983]. VLSI-algoritmen voor herkenning van context-vrije talen in lineaire tijd. Rapport IN 24/83, Stichting Mathematisch Centrum, Juni 1983.

Tomita, M. [1985]. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, 1985.

Tseytlin, G.E. and E.L. Yushchenko [1977]. Several aspects of theory of parametric models of languages and parallel syntactic analysis. In: *Methods of Algorithmic Language Implementation.* A. Ershov and C.H.A. Koster (eds.), Lect. Notes Comp. Sci. 47, Springer-Verlag, Berlin, 1977, 231-245.

Winograd, T. [1983]. *Language as a Cognitive Process.* Vol. 1: Syntax. Addison-Wesley Publishing Company, Reading, Mass., 1983.

Yonezawa, A. and I. Ohsawa [1988]. Object-oriented parallel parsing for context-free grammars. 1988, 773-778.

Zozel, M. [1973]. A parallel approach to compilation. In: *Principles of Programming Languages*, 1973, 59-70.

ABSTRACT OBJECTS AS ABSTRACT DATA TYPES REVISITED

by Alexander Ollongren

Department of Mathematics and Computer Science
Leiden University

PREFACE

The present paper is a revision of an article written by H. Gerstmann and the author and published in the Proceedings of the 1979 Copenhagen Winter School 'Abstract Software Specifications', Springer Lecture Notes in Computer Science No. 86. The main part of the paper is taken *verbatim* from the mentioned article (permission from the publisher has been requested); the last section is written anew.

I am happy that the compilers of the *Liber Amicorum* dedicated to Jaco de Bakker have decided to include the present paper, because it gives me the opportunity to discuss, albeit briefly, an issue which was considered in the seventies important in the field of the semantics of programming languages. The question which raised much attention at the time was whether the semantics should be defined in an operational manner or by different means. In the view of the school based in Vienna to which P. Lucas, K. Walk and many others belonged, the issue could be settled by defining a suitable formal universal interpreter. At the time I was much attracted to the idea and tried to contribute to it by formalizing the underlying so-called *objects*.
The interpreter was to be universal in the sense that it could give an interpretation to any program in any high-level programming language of the procedural type. In modern terminology the set of objects served as the basic data type of the interpreter. PL/I was given a semantics in this way [1], and in the seventies the Vienna group showed the usefulness of the method by formally defining a number of semantics and semantic concepts.

There was a rivalling school of thought, the Oxford school, led by ideas of Chr. Strachey, which set the scene for what is now called the denotational semantics of programming languages. When I first met Jaco de Bakker it was by no means clear which school would emerge as the most promising and I remember him saying that the future would tell. (In hindsight it is clear that both approaches were promising: the denotational method is well-founded and widely used today, the original idea of the universal interpreter defined in the Vienna Definition Language (VDL) led to the establishment of the Vienna Development Method (VDM), equally well-known). Since then Jaco wrote the magnificent volume on the semantics of programming languages [10], which is based on yet another concept: the set-theoretic approach.

By the end of the seventies the operational method of the Vienna school was under criticism from theoreticians who considered the basic objects in VDL 'not really abstract'. This was clearly an open invitation for one to try to counter the point and I presented at the mentioned Winter School an axiomatic treatment of the same - what could be more abstract? H. Gerstmann gave a lecture too, presenting an algebraic treatment. At the meeting we decided to cooperate and contribute to the proceedings with just one paper: the main sections are reproduced here. The last section is revised because I wish to connect with modern developments in VDM.

Finally I must remark here that B. Nordström [11] a few years later gave yet another foundation of the abstract objects in the regime of constructive mathematics, thus definitely settling the account in favor of the abstract nature of the datatype

and so of the universal interpreter. The complexity of the device, however, remains a drawback as far as application is concerned.

INTRODUCTION

In the Vienna Definition Language (VDL) [1,2,3] 'abstract objects' are provided to describe the state space of symbolic machines as well as the phrase structure of programs. Since the language is used as a meta language to define the operational semantics of programming languages, some authors [4] refuse to consider abstract objects as being abstract. However, whether semantics are denotational or operational is just a matter of the mapping from the syntactic to the semantic domain, and not of the meta language in which the mapping is described.

As a matter of fact, the authentic reference [1] makes a clear distinction between abstract objects and their representations, and in a paper by H. ZEMANEK [5] isomorphic representations are discussed. In pursuit of this design philosophy P. LUCAS introduced the notion of a 'software device', which seems to be the earliest instance of an abstract data type. His paper presented at the Second Courant Computer Symposium in 1970 [6] contains an equational presentation of a stack and an implementation for it.

The purpose of the present paper is to show that abstract objects are indeed abstract data types in the strict algebraic form of [4]. To reveal the structural properties of the data type Abstract Object, it is derived from the data type Set in a sequence of refinements by means of enrichment and functors [7]. This way of proceeding does not only yield the original VDL objects, but also its advanced descendants in form of new objects [8]. The paper concludes with an outline of how this abstract data type can be used as model for parts of META IV, the meta language in the Vienna Development Method (VDM) [9].

ALGEBRAIC PROVISIONS

A specification of an abstract data type consists of a set of sorts S, a Set of operation symbols Σ, and a set of equations E between the operation symbols. A Σ-algebra contains for each sort S a carrier

$$A_s$$

and for each operator symbol of type

$$s1s2 \ldots sn \rightarrow s$$

a function

$$\sigma_A : A_{s1} \times A_{s2} \times \ldots \times A_{sn} \rightarrow A_s \; .$$

The class of all Σ-algebras together with their homomorphisms can be regarded as a category. In this category the word algebra $T(\Sigma)$ of well-formed terms built up from the operator symbols is initial, whereas in the subcategory of Σ-algebras that satisfy the set of equations this property is taken over by the quotient algebra $T(\Sigma E)$ with respect to the congruence relation generated by E.

The class of all initial algebras in the subcategory defines the abstract data type of the given specification. Within this class the subclass of all word algebras, with

T(ΣE) as its representative, yields the abstract syntax, and any other initial algebra is a model of the data type.

During stepwise specification by enrichment, new operation symbols on the given sorts and equations between them are added to the specification of the data type such that existing congruence classes are maintained. If new sorts are required too, a functor can be defined between the respective categories which assures that each model of the extended specification is also a model of the original one.

STEPWISE SPECIFICATION OF DATA TYPE ABSTRACT OBJECT

Starting from a specification of Set the data type Abstract Object is developed in a chain of refinements. The reader is assumed to have a model of Set at his disposal which enables him to verify the following equational presentation:

Data type: Set
Sorts: el, set, bool

Operation Symbols
insert: el set → set
delete: el set → set
has: el set → bool
Ø: → set

Equations

insert (e,insert(e,s))	=	insert(e,s)	
insert(e1,insert(e2,s))	=	insert(e2,insert(e1,s))	e1≠e2
delete(e,delete(e,s))	=	delete(e,s)	
delete(e1,delete(e2,s))	=	delete(e2,delete(e1,s))	e1≠e2
insert(e,delete(e,s))	=	insert(e,s)	
delete(e,insert(e,s))	=	delete (e,s)	
insert(e1,delete(e2,s))	=	delete(e2,insert(e1,s))	e1≠e2
delete(e,Ø,)	=	Ø	
has(e,insert(e,s))	=	true	
has(e,delete(e,s))	=	false	
has(e1,insert(e2,s))	=	has(e1,s)	e1≠e2
has(e1,delete(e2,s))	=	has(e1,s)	e1≠e2 .

The first step of refinement introduces assumptions about the elements. Each element is again a set, prefixed by a name to identify it.

el = name set.

However, elements are not just considered Cartesian products, which would leave the axioms essentially unchanged. Since the first component is to act as a name, the occurrence of equal names requires special treatment. In this case the second component is overwritten by the insert and forgotten by the delete operation. This yields the

Data Type: Set of named Sets
Sorts: name, set, bool

Operations
insert: name set set → set
delete: name set set → set

```
has: name set set → bool
 ∅:               → set
```

```
Equations
insert (n,s1,insert(n,s2,s))     =  insert(n,s1,s)
insert (n1,s1,insert(n2,s2,s))   =  insert(n2,s2,insert(n1,s1,s))     n1≠n2
delete(n,s1,delete(n,s2,s))      =  delete(n,s1,s)
delete(n1,s1,delete(n2,s2,s))    =  delete(n2,s2,delete(n1,s1,s))     n1≠n2
insert(n,s1,delete(n,s2,s))      =  insert(n,s1,s)
delete(n,s1,insert(n,s2,s))      =  delete(n,s1,s)
insert(n1,s1,delete(n2,s2,s))    =  delete(n2,s2,insert(n1,s1,s))     n1≠n2
delete(n,s,∅)                    =  ∅
has(n,s1,insert(n,s1,s))         =  true
has(n,s1,delete(n,s1,s))         =  false
has(n1,s1,insert(n2,s2,s))       =  has(n1,s1,s)                      n1≠n2
has(n1,s1,delete(n2,s2,s))       =  has(n1,s1,s)                      n1≠n2
```

In order to show that the refined version is still a model of Set, a functor

F: Set of named Sets → Set

is introduced by the mappings

```
F(name set)   =  name
F(set)        =  set
F(bool)       =  bool
```

for objects and

```
F(insert(n,s1,s))    =    insert(n,s)
F(delete(n,s1,s))    =    delete(n,s)
F(has(n,s1,s))       =    has(n,s)
F( ∅)                =    ∅
```

for operations. With its help the original equations can be recovered from the refined ones, for instance

```
insert(n,s1,insert(n,s2,s))  =  insert(n,s1,s)
   F
------>
insert(n,insert(n,s))   =  insert(n,s)
```

or

```
(n1≠n2) ⇒
insert(n1,s1,insert(n2,s2,s))   =  insert(n2,s2,insert(n1,s1,s))
   F
----->
(n1≠n2) ⇒
insert(n1,insert(n2,s))=  insert(n2,insert(n1,s))   .
```

An inspection of the axioms discloses a striking symmetry between the operations of insertion and deletion. This property suggests the definition of one operation in terms of the other. Since, due to the inductive definition of set, the empty set can be used in an element, it is not wrong to try

```
delete(n,s1,s)  =  insert(n,∅,s).
```

It is easy to see that with this definition the axioms containing delete become special cases of the other ones and can be removed. The remaining equations are:

insert(n,s1,insert(n,s2,s))	=	insert(n,s1,s)	
insert(n1,s1,insert(n2,s2,s))	=	insert(n2,s2,insert(n1,s1,s))	n1≠n2
insert(n,∅,∅)	=	∅	
has(n,s1,insert(n,s1,s))	=	true	
has(n,s1,insert(n,∅,s))	=	false	
has(n1,s1,insert(n2,s2,s))	=	has(n1,s1,s)	n1≠n2 .

To obtain abstract objects, the derived operation

select: name set → set

defined by

select(n,s) = s1 if has(n,s1,s) = true
∅ otherwise

is introduced instead of has. Substitution into the equations for has yields the corresponding equations for select

select(n,insert(n,s1,s)) = s1

select(n1,insert(n2,s2)) = select(n1,s) n1≠n2.

The final step includes a transition to the more familiar notation

name	←	sel(ector)
set	←	obj(ect)
empty set (∅)	←	null object (Ω)
insert(x,y,y')	←	mk(y',x,y)
select(x,y)	←	s*l*(x,y)

and the specification of a basis from which the objects are constructed. Within sort obj a set of generators el, called elementary objects, is assumed, which, combined with selectors, yield objects. Additional axioms extend the operations to elementary objects. The result of all these measures is the

Data Type: Abstract Object
Sorts: obj, sel

Operation Symbols

mk :	obj sel obj	→	obj
s*l*:	sel obj	→	obj
:		→	obj

Equations:

mk(mk(o,s,o1),s,o2)	=	mk(o,s,o2)	sort (o)≠el v o1≠Ω
mk(mk(e,s,o1),s,Ω)	=	Ω	sort(e) = el
mk(mk(o,s1,o1),s2,o2)	=	mk(mk(o,s2,o2),s1,o1)	s1≠s2
mk(Ω,s,Ω)	=	Ω	
mk(e,s,Ω)	=	e	sort(e)=el
s*l*(s,mk(o,s,o1))	=	o1	
s*l*(s1,mk(o,s2,o2))	=	s*l*(s1,o)	s1≠s2 .

Actually this is not the specification of a data type but of a class of data types. The presentation contains the two basis types sel and el which still must be specified to obtain a specific data type within the class. Any data object within the class is called a new object in reference [8].

The original VDL objects [1] appear as a special case in which sel is specified as a monoid and el as a set of atoms. Monoid multiplication is identified with functional composition so that the additional axioms

$$sl(s1*s2,o) = sl(s1,sl(s2,o))$$
$$mk(o,s1*s2,o1) = mk(o,s2,mk(sl(s2,o),s1,o1))$$

hold. The consequences of these additional axioms will be discussed after a model for abstract objects is available in the next section.

STANDARD MODEL FOR ABSTRACT OBJECTS

To show the consistency of the final equations, a model is presented. The model is based on the assumptions that models for the basis types have already been given and therefore applies to any data type within the class of abstract objects. For this reason it is called standard model.

Before the model can be defined the syntax must be specified. The syntax of the abstract data type is defined by the word algebra T with

Carriers: T_{obj} , T_{sel}

Operations

$mk_T : T_{obj} \times T_{sel} \times T_{obj} \to T_{obj}$

$sl_T : T_{sel} \times T_{obj} \to T_{obj}$

$\Omega_T : \to T_{obj}$.

A term t in T_{obj} is generated from the basis terms T_{sel} and T_{el} by means of the productions:

$t \leftarrow \Omega \mid e \qquad e \in T_{el}, s \in T_{sel}, t \in T_{obj}$

$t \leftarrow sl(s,t) \mid mk(t,s,t)$.

The terms t are considered as words 't' on which the operations are defined by

mk('t','s','t2') = 'mk(t1,s,t2)'
sl('s','t') = 'sl(s,t)'.

The axioms induce a partition of the set of words into disjoint classes of equivalent words. For each class there is a unique representative: sl can be eliminated from a term applying the axioms for this function. Any nested mk-term can be reduced by the other axioms to an equivalent one in which all s are distinct and obey a certain order. Thus any class can either be represented by 'Ω', 'e', or by a word of the form

'$mk(mk(...mk(t_0,s_1,t_1), \ldots , s_{n-1},t_{n-1}),s_n,t_n)$'

in which t_0 is an elementary or null object, any other t again a term of the same kind, and

$s_i \neq s_k$ for $i \neq k$.

The specification of abstract objects in a chain of refinements starting from Set indicates that there must exist a set theoretic model. Instead of pairs, however, the notion of mappings as provided in META IV [9] is used because the associated operations are moreadequate for the present purpose.

The model is an algebra

$$0 = (0_{obj}, 0_{sel}; mk_0, sl_0, \Omega_0)$$

with

Carriers: $0_{obj}, 0_{sel}$

and

Operations

$mk_0 : 0_{obj} \times 0_{sel} \times 0_{obj} \rightarrow 0_{obj}$

$sl_0 : \quad 0_{sel} \times 0_{obj} \rightarrow 0_{obj}$

$\Omega_0: \quad \rightarrow 0_{obj}$.

Objects are mappings in the domain

$$0_{obj} =_m (0_{sel} \rightarrow (0_{gen} \mid 0_{obj}))$$

with 0_{sel} and 0_{el} assumed to be specified and the generic set is

$$0_{gen} = 0_{el} \cup \{\Omega_0\} \quad .$$

Partial mappings

$$[s \rightarrow o(s) \mid s \in Sel] \qquad Sel \subset 0_{sel}$$

are completed by the convention

$$[s \rightarrow \text{if } s \in Sel \text{ then } o(s) \text{ else } [\,] \mid s \in 0_{sel}] \quad .$$

[ ] denotes the empty mapping, which is also used to represent Ω_0. The other operations are defined by the expressions

$$mk_0(o,s,o1) = (o \in 0_{el} \wedge o1 = [\,] \rightarrow o,$$
$$o \in 0_{el} \wedge o1 \neq [\,] \rightarrow [s \rightarrow o1],$$
$$o \notin 0_{el} \rightarrow o + [s \rightarrow o1])$$

$$sl_0(s,o) = (o \in 0_{el} \rightarrow [\,],$$
$$o \notin 0_{el} \rightarrow o(s)).$$

Since for the basis types the existence of models is assumed, there is already an isomorphism between them and the respective terms in T:

$h('e') = e$ $\quad$ $'e' \in T_{el}, e \in 0_{el}$

$h('s') = s$ $\quad$ $'s' \in T_{sel}, s \in 0_{sel}$.

Next the nullary operations must be associated

$$h('\Omega') = [\ ].$$

This partial mapping can now be uniquely extended to the classes of equivalent words in T: For each representative

$$'mk(mk(\dots mk(t_0,s_1,t_1), \dots s_{n-1},t_{n-1}), s_n,t_n)'$$

of a class its image is defined recursively by

$$h('mk(t_A,s,t_B)') = h('t_A') \cup [h('s') \rightarrow h('t_B')].$$

It is not difficult to show that the axioms are satisfied by the model, for instance

$$sl(s,mk(o,s,o1)) = o1 \quad \text{for } o \notin 0_{el} :$$

$$sl_0(s,mk_0(o,s,o1)) = sl_0(s,o + [s \rightarrow o1])$$

$$= (o + [s \rightarrow o1])(s) = o1.$$

To show the initiality of T, let

$$A = (A_{obj}, A_{sel} ; M, S, 0)$$

be any algebra in the subcategory of algebras satisfying the axioms and

$$u: 0 \rightarrow A$$

uniquely defined by

$u(e) = a$ $\quad$ $e \in 0_{el}, a \in A_{el}$

$u(s) = r$ $\quad$ $s \in 0_{sel}, r \in A_{sel}$

$u([\ ]) = 0$

$u([si \rightarrow oi \mid si \in Sel]) = M(u([si \rightarrow oi \mid si \in Sel \setminus \{sk\}]),u(sk),u(ok)).$

Again the conditions for a homomorphism are only proved for a typical case. For

$si \in Sel$

$u(S(si,[sj \rightarrow oj \mid sj \in Sel])) = u(oi) =$ $\quad$ (by the first axiom for sl)

$S(u(si),M(u([sj \rightarrow oj \mid sj \in Sel \setminus \{si\}]),u(si,u(oi))) =$

$S(u(si),u([sj \rightarrow oj \mid sj \in Sel]))$

$u(M([sj \to oj \mid sj \in Sel], si, o')) =$

$u([sj \to oj \mid sj \in Sel \backslash \{si\}] \cup [si \to o']) =$

$M(u([sj \to oj \mid sj \in Sel \backslash \{si\}]), u(si, u(o')) =$ (by the first axiom for mk)

$M(M(u([sj \to oj \mid sj \in Sel \backslash \{si\}), u(si), u(oi)), u(si), u(o')) =$

$M(u([sj \to oj \mid sj \in Sel]), u(si), u(o')).$

To obtain VDL objects within the standard model 0_{sel} must be specified as monoid and the equations

$$o + [s1*s2 \to o12] = o + [s2 \to o(s2) + [s1 \to o12]]$$

$$o(s1*s2) = o(s2)(s1)$$

added. For the null object the first equation becomes

$$[s1*s2 \to o12] = [s2 \to [s1 \to o12]]$$

so that in any object with components

$[s1 \to o1]$, $[s2 \to o2]$, and $[s1*s2 \to o12]$

the identity

$$o2 = [s1 \to o12]$$

holds. If the equations are dropped new objects over a selector monoid are created in which the objects o1, o2, and o12 can be chosen arbitrarily.

On the other hand, paths through an object can be equated by imposing additional axioms on the monoid. Let

$$0_{sel} = (\{0,1\}; *)$$

be a monoid with two selectors satisfying the axiom

$$1^i 0^k = 0^k 1^i$$

In this case a two dimensional array A is obtained, in which an element A(i,k) can be retrieved along any path equivalent to the path $1^i 0^k$.

SOME FURTHER REMARKS

In the previous section de VDM meta language META IV (cf. the introduction) is briefly used to represent abstract objects as mappings. In the present section the attention is directed to the problem of providing interpretations for some parts of the meta language in the abstract data type developed.

To start with the set of elementary objects obviously can be represented by the basis of the data type, i.e. generators of the sort el. Further classes of trees are interpreted as classes of new objects (multi-level arrays) with some appropriate set of selectors, i.e. the sort sel. This allows an interpretation of an expression in the abstract syntax like

$$A :: s1: B1 \quad s2 : B2 \ldots sn : Bn \qquad n > 0$$

as the class of multi-level arrays characterized by

$$a \in A \Leftrightarrow s/(si,a) \in Bi \qquad i=1,2, \ldots ,n.$$

The symbol :: can be read as 'is composed of' ([12]), p. 122). An axiomatic treatment of (generalized) multi-level arrays is to be found in [13].

Elements of the class can be created under the META IV conventions by so-called make-functions. Let Bi (i = 1,2, ...,n) be given sets, possibly of trees. Then the function mk-A of type

$$\text{mk-A} : B1 \times B2 \times \ldots \times Bn \rightarrow A$$

evaluates (for a given sequence of selectors s1, s2, ..., sn) to an element for which the following property holds:

$$s/(si,\text{mk-A}\ (b1, \ldots, bn)) = bi \qquad i = 1,2 \ldots, n.$$

This property is the same one as the characterizing property of the expression we started out with. META IV allows to write instead of the last equality:

$$si(\text{mk-A}(b1, \ldots ,bn)) = bi$$

so that the selector si takes the role of a function of type $A \rightarrow Bi$ (i = 1,2,...,n).

Within the abstract data type the make-functions admit the following interpretation:

$$\text{mk-A}\ (b1, \ldots , bn) = a \quad \Leftrightarrow \quad a \in A \wedge a =$$

$$mk^{n}(^{n}\Omega,s1,b1),s2,b2),\ldots)sn,bn)$$

with

$$mk^{n}(^{n}o,s1,b1),s2,b2)\ldots),sn,bn) =$$

$$mk^{n-1}(^{n-1}mk(o,s1,b1),s2,b2),\ldots),sn,bn)$$

and

$$mk^{0}(o,) = o\ .$$

From this we see that mk-A can be expressed as a lambda form:

$$\text{mk-A} = \lambda\ (b1,b2,\ldots,bn).\ \ mk^{n}(^{n}\Omega\ ,s1,b1),s2,b2),\ldots),sn,bn)\ .$$

As before the sequence of selectors s1,...,sn is supposed to be given.

REFERENCES

[1] LUCAS, P. and WALK, K.: On the Formal Description of PL/I, Annual Review of Automatci Programming 6, 3(1969)

[2] WEGNER, P.: The Vienna Definition Language, ACM Comp. Surveys 4, 1(1972), 5-63

[3] OLLONGREN, A.: Definition of programming languages by interpreting automata, Acad. Press 1974.

[4] GOGUEN, J.A. et al.: Abstract Data Types as initial Algebras and the Correctness of Data Representations, SIGGRAPH Notices (May 1975), 89-93

[5] ZEMANEK, H.: Abstrakte Objekte, Elektronische Rechenanlagen 10, 5(1968), 208-217

[6] LUCAS, P.: On the Semantics of Programming Languages and oftware Devices, in Formal Semantics of Progr. Lang., edited by R. Rustin, Prentice Hall 1972, 41-57

[7] EHRIG, H. et al.: Stepwise Specification and Implementation of Abstract Data Types, in Automata, Languages and Programming, edited by G. Ausiello and C. Boehm, Lect. Notes in Comp. Sc. Vol. 62, Springer-Verlag 1978, 205-226

[8] GOEMAN, H.J.M. et al.: Axiomatiek van Datastructuren, MC Syllabus 37 Colloquium Capita Datastructuren (1978), 85-98

[9] BJöRNER, D. and JONES, C.B., The Vienna Development Method: The Meta-Language, Lect. Notes in Comp. Sc. Vol. 61, Springer-Verlag 1978

[10] DE BAKKER, J.W.: Mathematical Theory of Program Correctness, Prentice Hall, 1980

[11] NORDSTRöM, B. : Multilevel functions in Martin-Löf's type theory, Lect. Notes in Comp. Sc. Vol.217, 206-221, Springer Verlag 1986

[12] JONES, C.B.: Systematic Software Development Using VDM, Prentice Hall, 1986.

[13] BERGSTRA, J.A. et al.: Axioms for Multilevel Objects, Annales Societas Mathematicae Polonae Series IV: Fundamenta Informaticae, Vol. III,2, 171-180, (1979)

Uit het Zuiden komt deze Wind amici gevlogen,
Hij is uit de wereld van Database Theory getogen.
Ons raakvlak, mijn beste Jako, ligt vooralp,
in het organiseren van conferenties zoals ICALP.
Onze intersectie daarentegen vormen de Formele Aspecten,
die, vooral in het Zuiden, veel te weinig belangstelling wekten.

Jan Paredaens
Hoogleraar
Universiteit Antwerpen

What is in a Step

A. Pnueli, M. Shalev

Dept. of Applied Mathematics & Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

May 1988

This paper presents a proposal for the definition of a step in the execution of a statechart. The proposed semantics maintains the synchrony hypothesis, by which the system is infinitely faster than its environment, and can always finish computing its response before the next stimulus arrives. However, it corrects some inconsistencies present in previous definitions, by requiring global consistency of the step.

The research reported here has been partially supported by ESPRIT project 937 (DESCARTES) granted to AdCad Israel. The research of the second author was done as part of her M.Sc. thesis at the Weizmann Institute.

1. Introduction

The language of Statecharts has been proposed by D. Harel ([H]) as a visual language for the specification and modeling of *reactive systems*. While the (graphical) syntax of the language has been firmed up quite early, the definition of its formal semantics proved to be more difficult than originally expected. These difficulties may be explained as resulting from several requirements that seem to be desirable in a specification language for reactive systems, but yet may conflict with one another in some interpretations. Below, we list and shortly discuss each of these basic requirements.

To illustrate the discussed points we will use a restricted subset of the statechart syntax. The basic reaction of the system to external stimuli (events), are performed by ***transitions***. Transitions in the system are graphically represented by arrows connecting one state to another, and are labeled by a ***label*** which typically has the form e/a. In such a label, the ***event*** e ***triggers*** (enables) the transition, i.e., allows it to be taken. The optional ***action*** a is performed when the transition is actually taken. Typically, the action has the form $f!$, which means that it ***generates*** the event f, or may be an assignment, assigning a value to a variable. Two transitions may be ***parallel*** to one another (also referred to as ***orthogonal***), which means that they can be performed in the same step. Alternately, two transitions may be ***conflicting***, e.g., if they depart from the same state, and then, at most one of them can be taken at any given step.

Synchrony Hypothesis

One of the main requirements one may wish to associate with a specification language for reactive systems is the ***synchrony hypothesis***. This hypothesis assumes that the system is infinitely faster than the environment, and hence the response to an external stimulus is always generated in the same step that the stimulus is introduced. We may view this hypothesis as stating that the response is always simultaneous with the stimulus. We remind the reader that a long sequence of internal communications may be required to generate the outgoing response.

For example, we may have a set of parallel transitions with the following labels:

$$t_0 : a/e_1! \quad t_1 : e_1/e_2! \quad \ldots \quad t_n : e_n/b!$$

An incoming stimulus represented by the event a causes the transition t_0 to be activated and generate the event e_1. In turn, the transition t_1 responds to e_1 by generating e_2. This chain reaction continues until the transition t_n responds to e_n by generating b, which may be the final response of the system. According to the synchrony hypothesis b (and $e_1, \ldots e_n$) are all generated in the same step in which

the event a is presented to the system.

The synchrony hypothesis is an abstraction that limits the interference that may occur in the time period separating the stimulus and the response, and hence provides a guaranteed response as a primitive construct. In later stages of the development of the system, a more realistic modeling of the actual implementation can be done by introducing explicit delay elements if necessary.

Yet, Retaining Causality

In spite of the simultaneity abstraction, we should retain the distinction between cause and effect.

Consider for example the case that no external stimulus is given, and the system has two ready parallel transitions, with the following labels

$$t_1 : a/b! \quad t_2 : b/a!$$

In principle one may consider a semantics in which both transitions are taken while generating the events a and b. The justification for taking the transition t_1 with the trigger a is that a is generated by the transition t_2 at the same step. Similarly, the generation of b by t_1 justifies taking t_2.

This is a situation we like to exclude. The principle of causality requires that there is a clear causal ordering among the transitions taken in a step, such that no transition t relies for its activation on events generated by transitions appearing later than t in that ordering.

Expressing Priorities

An important feature of specification formalisms for real-time and reactive systems is the ability to assign *priorities* to responses.

Assume, for example, a system with two conflicting transitions with labels

$$t_1 : a \quad t_2 : b$$

We may consider t_1 to be a response to the event a while t_2 is a response to the event b. In the (probably infrequent) case that both a and b occur in the same step, the response is chosen non-deterministically. In many cases we may want to stipulate that, in the case both a and b occur, the response to b should have the higher priority. In Statecharts, this is expressed by using the *negation* of events. The general syntax of labels allows a *triggering expression* which is a boolean expression over events. The labels

$$t_1 : a \land \neg b \quad t_2 : b$$

ensure that if a or b occur exclusively, then as before, t_1 or t_2 are taken, respectively. However, if a and b occur together in the same step, then only t_2 is taken.

The three requirements listed above, i.e., *synchrony, causality*, and *priorities*, led to the semantics defined in [HPSS].

The basic approach presented in [HPSS] is that the behavior of a statechart is described as a sequence of *steps*, each step leading from one stable configuration to the next. The environment may introduce new external events at the beginning of each step. The response of the system to these input events is built up of a sequence of *micro-steps*. The first micro-step consists of all the transitions that are triggered by the input events. Subsequent micro-steps consist of all the transitions triggered by the set of events containing the input events, as well as all the events generated by previous micro-steps. Since there are only finitely many transitions that can be taken in a step, the sequence of micro-steps always terminates when there are no additional enabled transitions. This concludes a single step, and the set of events generated in any of the micro-steps is defined as the events generated during this step. It is not difficult to see that all the three requirements of *synchrony, causality* and expressing *priorities* via negations of events, are satisfied by the [HPSS] semantics. It also has the distinct advantage of being computationally feasible, which implies existence of an efficient implementation.

Unfortunately, the approach presented in [HPSS] has several deficiencies. The main ones are that it is highly operational, strongly depends on the ordering between micro-steps, and does not possess the property of *global consistency*. To illustrate the last point, consider two parallel transitions with the labels

$$t_1 : \neg a/b! \quad t_2 : b/a!$$

The semantics of [HPSS] constructs for this case the step $\{t_1, t_2\}$, even when there are no input events. This step generates the events $\{a, b\}$. We may complain that this step is not globally consistent, because it includes both the event a and the transition t_1, whose triggering condition requires that a is not generated during the current step. We refer to this phenomenon as *global* inconsistency, since the sequence of micro-steps, consisting of $\{t_1\}$ first, followed by $\{t_2\}$, is *locally* consistent. It is justified to take t_1 in the first micro-step and t_2 in the second micro-step, since they are both enabled at these points. It is only when we sum the effect of the complete sequence, that the inconsistency is discovered.

Well defined programming and specification languages usually possess two types of semantics. An operational semantics defines the behavior of a program or a specification in terms of a sequence of simple and atomic *operations*. It usually provides important guidelines for the implementor of the execution engine of such programs or specifications (compiler or interpreter). The other type of semantics is a *declarative* one, which bases the definition of the meaning of a program

on some kind of equational theory (using fixpoints for iteration and recursion), and attempts to ignore operational details such as order of execution, etc. We have intentionally avoided using the term *denotational* semantics which implies *compositionality* in addition to declarativity.

The declarative semantics, being based on simpler mathematical principles, is the one that underlies formal reasoning about programs and specifications, such as comparing two programs for equivalence or inclusion.

A proven sign of healthy and robust understanding of the meaning of a programming or a specification language is the posession of both an operational and declarative semantics, which are consistent with one another. Based on this criterion, our first attempt was to define a declarative semantics consistent with the operational semantics of [HPSS]. We soon found out that one of the main requisites for a declarative semantics is global consistency which is absent from [HPSS].

The present paper attempts to achieve the goal of assigning mutually consistent operational and declarative semantics to the specification language of Statecharts. For that purpose we had to impose some restrictions on the syntax of statecharts, and somewhat modify the operational approach presented in [HPSS].

As seen below, the declarative semantics is based on fixpoints as is often the case. However, in the case considered here, the situation is more complex because, due to the presence of negations, the basic operator is in general non-monotonic.

Anticipating the formal development below, the basic fixpoint equation associated with a step is

$$T = En(T)$$

In this equation, T is a set of transitions which are candidates for being taken together in a step. The function $En(T)$ yields the transitions which are enabled by the events generated by the set of transitions T. We can translate each of the requirements listed above into a condition on the solution T_a which is an acceptable set of transitions that can be jointly taken in a step.

- **Synchrony Hypothesis.** This requirement can be represented by the condition

  $$En(T_a) \subseteq T_a,$$

  which states that all the transitions enabled by the events generated by T_a are already in T_a. This implies that T_a is *maximal* in the sense that no additional transitions can be taken in this step.

- **Causality.** This is represented by the requirement of *inseparability* expressed by the requirement that there exists no $T \subset T_a$, such that

  $$En(T) \cap (T - T_a) = \phi$$

This means that if we try to stop at any subset T which is strictly contained in T_a, there is always an additional enabled transition in $T - T_a$ that can be added to T. In the usual case of monotonic operators this corresponds to *minimality*.

- **Expressing Priorities.** This is provided by allowing negations of events in the triggering expressions.
- **Global Consistency.** This is represented by the condition

$$T_a \subseteq En(T_a),$$

which states that each transition in T_a is enabled on the *complete* set of transitions T_a.

The main concerns considered in this paper are not unique to Statecharts, and have to be faced by any language intended for the specification of reactive systems, such as ESTEREL (see [BC], [BG]) and LUSTRE (see [BCH]). These two languages have also adopted the principles of *synchrony, causality* and *global consistency*. However, they avoid the complex interplay between non-determinism and priorities, present in Statecharts, by ruling out any program giving rise to these problems as illegal.

We should realize that this paper considers only the *micro-semantics* of statecharts, by defining the fine structure of a single step. Single steps can be combined into an operational semantics, following the treatment of [HPSS]. We refer the reader to [HGR], where a denotational semantics for some versions of the operational semantics is considered, and to [HG] for a comparative discussion of the different factors determining the semantics of a reactive language.

2. Syntax

In this section we present the syntax and semantics of statecharts. In addition, we define some notations that are used in the following sections. The syntax presented here is based on the syntax that was introduced in [HPSS] with necessary modifications due to the special approach we adopted here to the semantics of statecharts. The modifications provide further restrictions on the structure of the event expressions, so that the concavity property, introduced in a following section, will hold.

A Statechart is a structure

$$SC = (S, r, \rho, \theta, \delta, V, \Pi, T)$$

where

- S is a set of *states*, r is the *root* state, and ρ, θ and δ are functions which describe some relations between the states.
- V is a set of *variables*.
- Π is a set of *primitive events*.
- T is a set of *transitions*.

The detailed definitions of the structure of states and of the structure of the transitions follow.

States and Their Structure

The set of states S represents both basic states and composite-states which contain other states as substates. The *hierarchy* function ρ, the *type* function θ and the *default* function δ, represent the ancestry relations among states as follows.

The *hierarchy function* $\rho : S \rightarrow 2^S$ defines the direct descendants (substates) of each state. If $\rho(x) = \rho(y)$ then it is required that $x = y$. There exists a unique state $r \in S$ such that $\forall s \in S$, $r \notin \rho(s)$. This state r is the *root* of the statechart. A state s is called *basic* if $\rho(s) = \phi$. Otherwise it is called *composite*. We define ρ^*, ρ^+, the transitive closures of ρ, by:

$$\rho^* = \bigcup_{i \geq 0} \rho^i(s), \quad \rho^+ = \bigcup_{i \geq 1} \rho^i(s).$$

The *type function* $\theta : S \rightarrow \{\text{AND}, \text{XOR}\}$ is a partial function that assigns to each state its type, and identifies it as either an *or*-state or an *and*-state. If $\rho(s) \neq \phi$ and $\theta(s) = \text{XOR}$ then $\rho(s)$ is a *xor* decomposition of s, i.e., when the system is in the state s it is in one and only one of its immediate substates. If $\rho(s) \neq \phi$ and $\theta(s) = \text{AND}$ then $\rho(s)$ is an *and* decomposition of s, i.e., when the system is in the state s it is simultaneously in all of its immediate substates.

The *default function* $\delta : S \rightarrow 2^S$ defines for a composite state s a set of states which are contained in s. If $x \in \delta(s)$, then $x \in \rho^+(s)$, and $\delta(s)$ is the *default set* for s. The intended meaning of the default set $\delta(s)$, is that if some transition names s as its target, then on taking this transition we enter the default states $\delta(s)$, as well as s itself. The typical situation is that s is an *or*-state and $\delta(s)$ is a singleton. In the case that $|\delta(s)| > 1$, a non-deterministic choice is implied. For simplicity, we assume that $|\delta(s)| = 1$ for each *or*-state s.

In this paper we do not consider *history symbols H* which are discussed in [HPSS]. This detail is not essential for the understanding of the approach we present, and using the functions that were defined in [HPSS] we need only slight modifications to add history to the syntax and the semantics considered here.

Terms

The set of terms $\mathcal{T}$ is defined by

1. If $n \in N$ is a numeral, then $n \in \mathcal{T}$.
2. If $v \in V$ is a variable, then both v, *new* $(v) \in \mathcal{T}$.
3. If *op* is a k-ary operation, and $t_1, \ldots, t_k \in \mathcal{T}$, then *op* $(t_1, \ldots, t_k) \in \mathcal{T}$.

In general we allow an arbitrary number of data domains over which we allow constants, typed variables and terms. For simplicity we consider here only the data domains of the integers, and of the booleans, which are considered next.

The notation *new* (v) refers to the newly assigned value in the current step. Thus, the test *new* $(x) = x + 1$ checks whether the value of x at the end of the step is greater by one than its value in the beginning of the step.

Boolean Terms

The set of boolean terms B is defined by

1. *true* , *false* $\in B$.
2. If $s \in S$ is a state, then *in* $(s) \in B$.
3. If $t_1, t_2 \in \tau$, then $(tRt_2) \in B$ for each $R \in \{=, >, <, \neq, \leq, \geq\}$.
4. If $b, b_1, b_2 \in B$ are boolean terms, then $\neg b$, $b_1 \vee b_2$, $b_1 \wedge b_2 \in B$.

Boolean terms are expressions that should evaluate to truth values. The expression *in* (s) is true if currently state s is active, i.e., the system is in state s. A boolean term which does not contain a subterm of the form *new* (v) is called an *old boolean term*.

Event Expressions

Event expressions are similar to boolean terms, in testing whether certain conditions hold in the current configuration, and yielding a boolean value as a result. However, while boolean terms are restricted to the examination of variables (either in their old or in their new version), event expressions can also test for the presence or absence of events.

We define the sets of *positive event expressions* E^+ and *negative event expressions* E^-. These two sets are then combined to form the set of *event expressions* E.

Positive Event Expressions E^+

1. The null event, $\lambda \in E^+$.

2. The primitive events $\Pi \subseteq E^+$.
3. If e_1, e_2 are positive event expressions, then so are $e_1 \wedge e_2$, and $e_1 \vee e_2$.
4. If $s \in S$ is a state, then *entered*(s), *exited*$(s) \in E^+$.
5. If $v \in V$ is a variable, then *assigned*$(v) \in E^+$.
6. If $e \in E^-$ is a negative event expression, then $\neg e \in E^+$.

The intended meaning of the positive event expressions, are that these expressions only become "more true" as we add more events to the set of events present in the current step. This means that they can only change from $\perp$ (undefined) to **T**, or from **F** to **T**, but never from **T** to **F**. The expression $e \in \Pi$ tests for the presence of the event e in the current step. The particular case of λ, tests for the presence of the null event in the current step. By definition, the test for the null event is always true.

By clause 3, any ***positive*** boolean combination of positive event expressions is also positive. The special events *entered*(s) and *exited*(s) are considered to occur as soon as a transition, which respectively enters or exits the state s, is taken.

The event *assigned*(v) is caused by the execution of an action which assigns a value to the variable v. This action also causes the term *new* (v) to be defined.

Negative Event Expressions

1. If $e \in E^+$ is a positive event expression, then $\neg e \in E^-$.
2. If $e_1, e_2 \in E^-$ are negative event expressions, then so are $e_1 \wedge e_2$, and $e_1 \vee e_2$.

Negative event expressions are intended to capture those expressions that can become "less true" as more events are generated. They can only change from $\perp$ to **F** or from **T** to **F**.

The expression $\neg e$ tests for the absence of the event e in the current step. Any positive boolean combination of negative events yields a negative event.

Event Expressions

1. $E^+ \cup E^- \subseteq E$.
2. If $e_1, e_2 \in E$ are event expressions then so are $e_1 \wedge e_2$, and $e_1[c]$, where c is a boolean term.

The event $e[c]$ is caused whenever e happens while the condition c is true. It can be viewed as the conjunction $e \wedge c$, requiring both e and c to hold.

Note that the class of event expressions is closed under conjunction but not under disjunction or negation. This shows that an event expression is either a

positive expression, or a negative expression, or a conjunction of a positive and a negative expression. Thus, $e_1 \wedge (\neg e_2)$ is an admissible event expression, but $e_1 \vee (\neg e_2)$ is not.

Actions

The set of actions A is defined inductively as follows:

1. The *null* action. $\epsilon \in A$.
2. If $e \in \Pi$ is a primitive event, then $e!$ is an action.
3. If $u \in V$ is a variable and t is an old term of compatible type, then $u := t$ is an action, to which we refer as an *assignment*.
4. If a_1, a_2 are actions then so is (a_1, a_2), provided a_1 and a_2 do not contain assignments to the same variable.

The actions are the instantaneous responses of the system to the external stimuli in addition to the internal change of state. They include generation of events (2), assignment to variables (3). Actions can be combined into sets of actions (4).

3. Orthogonal Sets and Configurations

We introduce here some notations and definitions from [HPSS] that are used in the following.

States

- We define *Basic* $\subseteq S$ to be the set of basic states, i.e., states s such that $\rho(s) = \phi$.
- For a set of states X, the *Lowest Common Ancestor* of X, denoted by $lca(X)$ is defined to be the state x such that
  (a) $X \subseteq \rho^*(x)$.
  (b) $\forall s \in S, \quad X \subseteq \rho^*(s) \Rightarrow x \in \rho^*(s)$
- For a set of states X, the *strict Lowest Common* OR-*Ancestor* of X, denoted by $lca^+(X)$ is defined to be the state x satisfying
  (a) $X \subseteq \rho^+(x)$.
  (b) $\theta(x) = \text{OR}$
  (c) $\forall s \in S \quad [\theta(s) = \text{OR}\ ,\ X \subseteq \rho^+(s)] \Rightarrow x \in \rho^+(s)$

- Two states x, y are *orthogonal*, denoted by $x \perp y$, if either $x = y$ or their *lca* is an AND state, that is, $\theta(lca(\{x, y\})) =$ AND.
- A set of states X, is an *orthogonal set* if for every $x, y \in X$, $x \perp y$. Note that the singleton set $\{x\}$ is always an *orthogonal* set. In the more general case, not considered here, that $|\delta(s)| > 1$, it is required that $\delta(s)$ be an orthogonal set.
- A set X is *an orthogonal set relative to* $s \in S$ if:
  (a) $X \subseteq \rho^*(s)$.
  (b) X is an orthogonal set.
- A set X is *a maximal orthogonal set relative to* $s \in S$ if:
  (a) X is an orthogonal set relative to s.
  (b) $\forall y \in \rho^*(s), y \notin X \Rightarrow X \cup \{y\}$ is not orthogonal.

  Note that $\forall s \in S$, $\{s\}$ is a maximal orthogonal set relative to s.

Upwards and Downwards Closures

Given a set of states X, we define the set X' to be the *upwards closure* of X if it is the *smallest* set satisfying the following conditions:

- $X \subseteq X'$
- The ancestor of any state in X' is also in X', i.e., if $\rho(s) \cap X' \neq \phi$, then $s \in X'$.

We denote the upwards closure of a state set X by $up(X)$. A set X is defined to be *upwards closed* if $X = up(X)$.

The *downwards closure* of a set X is defined to be the *smallest* set X' satisfying the conditions:

- $X \subseteq X'$
- For each composite *and*-state $s \in X'$, $\rho(s) \subseteq X'$, i.e., X' contains *all* the descendants of s.
- For each composite *or*-state $s \in X'$, $\delta(s) \in X'$, i.e., X' contains the default descendant of s.

We denote the downwards closure of a state set X by $down(X)$. A set S is defined to be *downwards closed* if $X = down(X)$.

Labels

The set of labels L is the set of pairs $E \times A$. For $l = (e, a)$ we write e/a. Informally, if e/a is a label of a transition t, then t is triggered by e and a is executed when t is taken.

Transitions

The set of transitions T given by a set of triples $2^S \times L \times 2^S$. A transition $t = (X, l, Y)$ consists of a *source set* X, a *target set* Y and a label l, where X and Y are orthogonal sets of states. Informally, if $l = e/a$, the system is in X and e occurs, then t is enabled and can be taken. If t is taken then a is executed and the system is then at Y. We denote the sets X and Y by $source(t)$ and $target(t)$ respectively.

Configurations

- A *state configuration* of $s \in S$ is an orthogonal set relative to s, all of whose members are basic states.
- A *maximal state configuration* X of $s \in S$ is a maximal orthogonal set, relative to s, all of whose members are basic states. In the case s is the root r, we refer to X simply as a maximal state configuration.
- A *store* St is a mapping from all the variables to values. The mapping can be partial and then we say that the value of the variable is $\perp$ interpreted as being undefined.

  Let Nat be the natural numbers domain and let $Bool$ be the boolean domain. A *partial store* is a function $St : V \rightarrow (Nat + Bool)_{\perp}$. A *total store* is $St : V \rightarrow Nat + Bool$.
- A *system configuration* $C = (S, St)$ consists of S, a maximal state configuration, and a total store St.
- Without loss of generality, we may assume that the target set $target(t)$ of each transition t consists of (orthogonal) basic states. In case it is not so originally, we can always replace Y by $down(Y) \cap Basic$, which will obey the simplifying restriction without changing the bahavior of the statechart.
- The *initial state configuration* X_0, is defined to be $down(\{r\}) \cap Basic$ where r is the root state.

Relations Between Transitions

- For a transition $t \in T$, we define the *arena* of t to be the lca^+ of the source and target of t. That is if $t = (X, l, Y)$ then $arena(t) = lca^+(X \cup Y)$.

Graphically, the arena of t is the lowest OR-state which fully contains the arrow representing t.

- Two different transitions $t_1 = (X_1, e_1/a_1, Y_1)$ and $t_2 = (X_2, e_2/a_2, Y_2)$ are said to be ***structurally consistent*** if:

  (a) $arena(t_1) \perp arena(t_2)$.

  (b) The set $a_1 \cup a_2$ contains at most one assignment to each variable $v \in V$.

  Otherwise, t_1 and t_2 are said to be ***structurally conflicting***.

  Note that every transition is structurally consistent with itself.

- A set T of transitions is said to be a ***structurally consistent set*** if $\forall t_1, t_2 \in T$, t_1 and t_2 are structurally consistent.

- A transition $t = (X, e/a, Y)$ is ***structurally relevant*** to a state configuration S, if $\forall x \in X\ \rho^*(x) \cap S \neq \phi$. An equivalent statement of the same fact is $X \subseteq up(S)$.

4. The Evaluation Function m

We introduce here the evaluation function of terms, boolean terms, and event expressions.

Additional Notations

Let A be a consistent set of assignments. We consider here a more general case than the one allowed in the restricted statechart syntax. The restricted syntax requires that the term t appearing in the assignment $x := t$ is an *old* term, i.e., contains no subterms of the form $new(y)$. The definition below covers the case that t may contain new subterms.

We can associate with A a set of equations over $V \cup V'$ (where $V' = \{x' \mid x \in V\}$), denoted by $Eq(A)$, by generating for each assignment $a : x := t$ an equation $eq(a) : x' := t'$. The right hand side t' of this equation, is obtained from t by replacing each occurrence of $new(y)$ by y'.

Let $St: V \rightarrow nat + Bool$ be a total store, and $St': V' \rightarrow (Nat + Bool)_\perp$, a partial store. We say that the pair (St, St') ***satisfies*** the set of assignments A, if the equations Eq (A) are satisfied over the interpretation (St, St'), i.e.,

$$(St, St') \models \text{Eq (A)}.$$

This means that when we evaluate both sides of each equation $x' = t'$, using the values provided by (St, St'), they evaluate to equal values, possibly $\perp$.

We say that a set of assignments A is *consistent* over a store St, if there exists some partial store St', such that (St, St') satisfies A. For the restricted case that all terms are old, we have that $t' = t$, and hence a sufficient condition for consistency is that, for each variable x, A contains at most one assignment assigning values to x.

Given a total store $St : V \rightarrow Nat{+}Bool$, and a set of assignments A consistent over St, we define $new_store(St, A)$ to be the store $St' : V' \rightarrow (Nat{+}Bool)_\perp$ which is the *minimal* store of that type, such that (St, St') satisfies A.

Example : Let

$$A = \{x := 5 + y,\ y := new(u) + 1,\ u := new(y) - 1\}$$

then,

$$Eq(A) = \{x' := 5 + y,\ y' := u' + 1,\ u' := y' - 1\}$$

consider a store St such that $St[y] = 0$. There are many solutions to $(St, St') \models Eq(A)$. For example, both $St'_0 : \{x' : 5,\ y' : 1,\ u' : 0\}$ and $St'_1 : \{x' : 5,\ y' : 2,\ u' : 1\}$ are solutions. However, there exists only one minimal solution, which for the above case is:

$$new_store(St, A) = St'_{min} : \{x' : 5,\ y' : \perp,\ u' : \perp\}.$$

- Let T be a set of transitions. We denote by *assgn*(T) the set of all assignments appearing in the action part of the transitions in T. We denote by $Ev(T)$ the set of all events *generated* by actions of the transitions in T of the form $e!$

We now define the evaluation function m corresponding to $K = (C, T) = (S, St, T)$, where C is a system configuration and T is a set of transitions. We refer to K as an *extended configuration*. An extended configuration represents an intermediate situation, where we have already decided to take the transitions in T, starting from $C = (S, St)$. The set T defines the set $Ev(T)$ of events generated by T, and a partial new store, generated by the assignments in *assgn*(T).

Evaluation of Terms

$m_\tau : \mathcal{T} \rightarrow K \rightarrow Nat_\perp$

- For a numeral $n \in N$,

$$m[\![n]\!](K) = nat(n)$$

i.e., the arithmetical value denoted by the numeral.

- For a variable $v \in V$,

$$m[\![v]\!](K) = St[v].$$

- For a variable $v \in V$,

$$m[\![new(v)]\!](K) = St'[v'],$$

where $St' = new_store(St, assgn(T))$. Note that $m[\![new(v)]\!](K)$ may yield $\perp$.

- If t is a term and op is a unary algebraic operation, then

$$m[\![op(t)]\!](K) = \mathbf{op}(m[\![t]\!])(K),$$

where **op** is the semantic operation corresponding to op.

- If t_1, t_2 are terms and op is a binary algebraic operation, then

$$m[\![op(t_1, t_2)]\!](K) = \mathbf{op}(m[\![t_1]\!](K), m[\![t_2]\!](K)),$$

where **op** is the semantic operation corresponding to op.

All the arithmetic operations are assumed to be *strict* with respect to their arguments.

Evaluation of Boolean Terms

$m_B : B \to K \to Bool_\perp$

- $m[\![true]\!] = \mathbf{T}$, $m[\![false]\!] = \mathbf{F}$.
- If $s \in S$ is a state, then

$$m[\![in(s)]\!](K) = if\ \rho^*(s) \cap S \neq \phi\ then\ \mathbf{T}\ else\ \mathbf{F}.$$

- If $t_1, t_2 \in \tau$ are terms and $R \in \{=, <, >, \neq, \leq, \geq\}$ is a relation, then

$$m[\![(t_1\ R\ t_2)]\!](K) = if\ (m[\![t_1]\!](K)\ R\ m[\![t_2]\!](K))\ then\ \mathbf{T}\ else\ \mathbf{F}.$$

It is assumed that all relations R are strict.

- If $b \in B$ is a boolean terms, then

$$m[\neg b](K) = \neg(m[b](K)),$$

where for $b \in Bool_\perp$,

b	$\neg b$
T	**F**
F	**T**
$\perp$	$\perp$

- If $b_1, b_2 \in B$ are boolean terms, then

$$m[b_1 \wedge b_2](K) = m[b_1](K) \wedge m[b_2](K)$$

$$m[b_1 \vee b_2](K) = m[b_1](K) \vee m[b_2](K),$$

where for $b_1, b_2 \in Bool_\perp$

b_1	b_2	$b_1 \wedge b_2 = b_2 \wedge b_1$
–	**F**	**F**
T	$\perp$	$\perp$
T	**T**	**T**

b_1	b_2	$b_1 \vee b_2 = b_2 \vee b_1$
–	**T**	**T**
F	$\perp$	$\perp$
F	**F**	**F**

Note that the boolean operations of conjunction and disjunction are not strict.

Evaluation of Event Expressions

$m_E : E \rightarrow K \rightarrow Bool_\perp$

- $m[\lambda](K) = \mathbf{T}$.
- If $s \in S$ is a state, then

$m[entered(s)](K) = \mathbf{T}$ *iff* for some $t \in T, s \in \rho^+(arena(t))$, and $\rho^*(s) \cap target(t) \neq \phi$

$m[exited(s)](K) = \mathbf{T}$ *iff* $\rho^*(s) \cap S \neq \phi$, and for some $t \in T$, $s \in \rho^+(arena(t))$

This definition says that a transition t generates the event *entered*(s) if s is a state (strictly) contained in the arena of t, and contains some states in the target of t. The transition t generates the event *exited*(s) if s is currently on (has some basic descendants in S), and is strictly contained in the arena of t. Thus if s is any state, and t a self-loop transition, connecting s to itself, i.e., $source(t) = target(t) = \{s\}$, then t generates both the events *entered*(s) and *exited*(s).

- If $x \in V$ is a variable, then

$$m[assigned(x)]\!](K) = \mathbf{T} \text{ \textit{iff} there exists some assignment } [x := t] \in assgn(T).$$

- For $e \in \Pi$ a primitive event expression,

$$m[e](K) = \mathbf{T} \text{ \textit{iff} } e \in Ev(T).$$

- For e, e_1 and e_2 event expressions,

$$m[e_1 \wedge e_2](K) = m[e_1](K) \bigwedge m[e_2](K).$$

$$m[e_1 \vee e_2](K) = m[e_1](K) \bigvee m[e_2](K).$$

$$m[\neg e](K) = \neg m[e](K).$$

- If c is a boolean term, then

$$m[e[c]](K) = m[e](K) \bigwedge m[c](K).$$

where $\neg$, $\bigwedge$ and $\bigvee$ are the semantic boolean operators that were used in the evaluation of boolean terms.

Enabling A Transition

Let $K = (C, T) = (S, St, T)$, where $C = (S, St)$ is a system configuration and T is a set of transitions. Let $t = (X, e/a, Y)$ be a transition. We say that (C, T) *enables* t if:

(a) t is structurally relevant to S, i.e., $X \subseteq up(S)$.

(b) t is structurally consistent with every $t' \in T$.

(c) $m[e](C, T) = \mathbf{T}$.

We define $En(C, T) = \{t \mid (C, T) \textit{ enables } t\}$.

5. The Concavity Property

In this section we formulate the notion of concavity and show that the restrictions imposed on the syntax of statecharts guarantee that all the transitions are concave. The concavity property is a property on transitions, which states that a transition can not be enabled, disabled, and then enabled again as a result of adding more transitions to the extended configuration. The concavity property holds because the syntax of event expressions is restricted in a way that, once an event expression e evaluates to $\mathbf{T}$ in some phase of the step construction, and then changes to a different value at a later phase ($\mathbf{F}$ or $\perp$), it can never regain the value of $\mathbf{T}$. Concavity is essential for the main result of this paper, which is the equivalence of the operational and declarative definitions of a step, and may be viewed as a weaker version of monotonicity.

5.1 Properties of Terms, Boolean Terms and Event Expressions

Lemma 1 : Given a term t and a set of transitions T, then if $m[t](C,T) \neq \perp$ then $T \subseteq T' \Rightarrow m[t](C,T')=m[t](C,T)$ where m is the evaluation function defined above.

Proof : trivial - by structural induction.

This shows that terms are monotone with respect to the argument T.

Definition : We say that an event expression e (a boolean term c) *follows* a path $\pi = (b_1, b_2, ..., b_n)$ where $\forall i \in [1,n],\ b_i \in Bool_\perp$, if there exist sets of transitions $T_1, T_2, \ldots, T_n$ such that $T_1 \subseteq T_2 \subseteq \ldots \subseteq T_n$, $m[e](C,T_i)=b_i$ ($m[c](C,T_i)=b_i$), for $i = 1,\ldots,n$. We will represent a set of paths by a regular expression over the set $\{\mathbf{T},\mathbf{F},\perp\}$.

Lemma 2 : A boolean term can only follow paths in $\perp^*\mathbf{T}^* + \perp^*\mathbf{F}^*$.

Proof : The proof is by induction on the structure of the boolean term.
Base case :

- *true* passes only through paths in $\mathbf{T}^*$.
- *false* passes only through paths in $\mathbf{F}^*$.
- $in(s)$, where s is a state, passes only through the paths $\mathbf{T}^* + \mathbf{F}^*$.

- If t_1, t_2 are terms, then $(t_1 R t_2)$ for $R \in \{=, >, <, \neq, \leq, \geq\}$ passes only through paths in $\bot^*\mathbf{T}^* + \bot^*\mathbf{F}^*$, because R is strict and by lemma 1.

Inductive step : We assume that b, b_1 and b_2 only follow paths in $\bot^*\mathbf{T}^* + \bot^*\mathbf{F}^*$. It is trivial to show that $\neg b$ also can follow only these paths. We show that the lemma holds also for $(b_1 \vee b_2), (b_1 \wedge b_2)$.
There are three subcases to consider :

1. b_1 follows $\bot^{i_1}\mathbf{T}^{j_1}$, b_2 follows $\bot^{i_2}\mathbf{T}^{j_2}$, and w.l.o.g. we assume that $i_1 \leq i_2$:
   - $(b_1 \wedge b_2)$ follows $\bot^{i_2}\mathbf{T}^{j_2}$.
   - $(b_1 \vee b_2)$ follows $\bot^{i_1}\mathbf{T}^{j_1}$.
2. b_1 follows $\bot^{i_1}\mathbf{T}^{j_1}$, and b_2 follows $\bot^{i_2}\mathbf{F}^{j_2}$. Then,
   - $(b_1 \wedge b_2)$ follows $\bot^{i_2}\mathbf{F}^{j_2}$.
   - $(b_1 \vee b_2)$ follows $\bot^{i_1}\mathbf{T}^{j_1}$.
3. b_1 follows $\bot^{i_1}\mathbf{F}^{j_1}$, and b_2 follows $\bot^{i_2}\mathbf{F}^{j_2}$, and w.l.o.g. we assume that $i_1 \leq i_2$:
   - $(b_1 \wedge b_2)$ follows $\bot^{i_1}\mathbf{F}^{j_1}$.
   - $(b_1 \vee b_2)$ follows $\bot^{i_2}\mathbf{F}^{j_2}$.

Lemma 3 : A positive event expression passes only through paths in $\mathbf{F}^*\mathbf{T}^*$, and a simple negative event expression passes only through paths in $\mathbf{T}^*\mathbf{F}^*$.

Proof : The proof is by induction on the structure of the positive and negative event expressions.
Base case :

- The positive event expression λ always follows the path $\mathbf{T}^*$.
- The primitive events $e \in \Pi$, and the events *entered*(s), *exited*(s), *assigned*(v), become true as soon as T_i contains a transition that generates them. Since the set T_i is monotonically increasing with i, these events obviously follow a path in $\mathbf{F}^*\mathbf{T}^*$.

Inductive step : We assume that the lemma holds for e, e_1 and e_2 and show that it holds also for $(e_1 \vee e_2), (e_1 \wedge e_2)$ and $\neg e$.

- The proof for $\neg e$ is trivial.

- In the case of positive event expressions, assume that e_1 passes through $\mathbf{F}^{i_1}\mathbf{T}^{j_1}$, e_2 passes through $\mathbf{F}^{i_2}\mathbf{T}^{j_2}$ and that w.l.o.g. $i_1 \leq i_2$. Then it follows that
  (a) $(e_1 \wedge e_2)$ passes through $\mathbf{F}^{i_2}\mathbf{T}^{j_2}$.
  (b) $(e_1 \vee e_2)$ passes through $\mathbf{F}^{i_1}\mathbf{T}^{j_1}$.
- In the case of negative event expressions, assume that e_1 passes through $\mathbf{T}^{i_1}\mathbf{F}^{j_1}$, e_2 passes through $\mathbf{T}^{i_2}\mathbf{F}^{j_2}$ and that w.l.o.g. $i_1 \leq i_2$. Then it follows that
  (a) $(e_1 \wedge e_2)$ passes through $\mathbf{T}^{i_1}\mathbf{F}^{j_1}$.
  (b) $(e_1 \vee e_2)$ passes through $\mathbf{T}^{i_2}\mathbf{F}^{j_2}$.

Lemma 4 : An event expression passes only through paths in $(\bot + \mathbf{F})^*\mathbf{T}^*\mathbf{F}^*$.

Proof : The proof is by induction on the structure of the event expression.
Base case : For the case of event expressions which are either positive or negative event expressions, the lemma follows from lemma 3 and the obvious inclusions

$$\mathbf{F}^*\mathbf{T}^* \subseteq (\bot + \mathbf{F})^*\mathbf{T}^*\mathbf{F}^*$$
$$\mathbf{T}^*\mathbf{F}^* \subseteq (\bot + \mathbf{F})^*\mathbf{T}^*\mathbf{F}^*$$

Inductive step : We assume that the lemma holds for e, e_1 and e_2 and show that it holds also for $(e_1 \wedge e_2)$ and $e[c]$, where c is a boolean term. By lemma 2 and because $\bot^*\mathbf{T}^* + \bot^*\mathbf{F}^* \subseteq (\bot + \mathbf{F})^*\mathbf{T}^*\mathbf{F}^*$, it is sufficient to show that the lemma holds for $(e_1 \wedge e_2)$.
Assume that e_1 passes through a path in $(\bot + \mathbf{F})^{i_1}\mathbf{T}^{j_1}\mathbf{F}^*$, e_2 passes through a path in $(\bot + \mathbf{F})^{i_2}\mathbf{T}^{j_2}\mathbf{F}^*$ and let $i_3 = max(i_1, i_2)$ and $j_3 = min(j_1 + i_1, j_2 + i_2)$. Then

- If $i_3 \leq j_3$ then $(e_1 \wedge e_2)$ passes through a path in $(\bot + \mathbf{F})^{i_3}\mathbf{T}^{j_3 - i_3}\mathbf{F}^*$.
- If $i_3 > j_3$ then $(e_1 \wedge e_2)$ passes through a path in $(\bot + \mathbf{F})^{j_3}\mathbf{F}^*$.

5.2 The Concavity Property

Theorem *(The Concavity Property)* : given a system configuration C, if T_1, T_2 and T_3 are three sets of transitions such that $T_1 \subseteq T_2 \subseteq T_3$, then there is no transition t such that $t \in En(C, T_1), t \notin En(C, T_2)$ and $t \in En(C, T_3)$.

Proof : Assume to the contrary that t is a transition such that $t \in En(C, T_1), t \notin En(C, T_2)$, and $t \in En(C, T_3)$. Let $t = (X, e/a, Y)$, and $C = (S, St)$. Then $t \notin En(C, T_2)$ only if one of the following holds

1. $\exists x \in X \ \rho^*(x) \cap S = \phi$. But then, we have a contradiction to the fact that $t \in En(C, T_1)$.
2. $\exists t' \in T_2$ such that t and t' are in a structural conflict. But then, because $t' \in T_3$ it cannot be that $t \in En(C, T_3)$.
3. $m[e](C, T_2) \neq \mathbf{T}$. But then because $m[e](C, T_1) = \mathbf{T}$ and $m[e](C, T_3) = \mathbf{T}$, we have that e passes through $(\mathbf{T}, \mathbf{F}, \mathbf{T})$ or through $(\mathbf{T}, \perp, \mathbf{T})$ which contradicts lemma 4.

6. Definition of a Step

In this section we introduce two definitions of the sets of transitions that are considered to be admissible steps from a given system configuration. The first definition is declarative and is based on particular solutions of a fixpoint equation. The second definition is constructive and suggests an algorithm for computing all the possible steps. We then show that under the property of concavity, which the syntax guarantees, the two definitions coincide.

Separable Sets

A set of transitions T is defined to be separable if there exists a subset $T' \subset T$, such that

$$En(C, T') \cap (T - T') = \phi.$$

A set T is called *inseparable* if it is not separable.

A Declarative Definition

A set of transitions T which is inseparable and satisfies the equation

$$T = En(C, T)$$

is called an *admissible step* of the system from configuration C.

We refer to such a set also as an *inseparable solution.*

The *Yield* of a Step

Let C be a configuration and T an admissible step from C. We define the configuration following the application of the step T to the configuration C, as follows:

$$Next\colon Config \times 2^{Trans} \rightarrow Config.$$

$$Next((S, St), T) = (S', St')$$

where

$$S' = \left(S - \bigcup_{t \in T} \rho^*(arena^+(t))\right) \cup \left(\bigcup_{t \in T} target(t)\right)$$

and

$$St' = update(St, new_store(St, assgn(T))).$$

Given two stores St and St', the function *update* (St, St') yields a store St'' defined by

$$St''[v] = \ \mathit{if}\ St'[v] = \bot\ \mathit{then}\ St[v]\ \mathit{else}\ St'[v]$$

A Constructive Definition

The definition presented above for admissible steps is declarative. An alternative definition can be given by a non-deterministic procedure that builds a step T by adding one transition at a time.

1. Initially $T = \phi$.
2. Compare $En(C, T)$ to T.
   (a) If $T = En(C, T)$ terminate and report success.
   (b) If $T \subset En(C, T)$, pick a transition $t \in En(C, T) - T$ and add it to T. Repeat step 2.
   (c) Otherwise, i.e., $T \not\subseteq En(C, T)$, report failure.

Proposition : A set of transitions T is an inseparable solution to the equation $T = En(C, T)$ iff it can be constructed by the above procedure.

Proof : Assume that T_0 is an inseparable solution to the equation $T_0 = En(C, T_0)$.

We apply the procedure as specified, with the modification that the only new transitions to be added to T are picked from $(En(C, T) - T) \cap T_0$. We show:

- The procedure cannot fail.
  The procedure can fail only by having $T \subset T_0$, $T \subset En(C,T)$, $t \in (En(C,T) - T) \cap T_0$ and $t' \in T \cup \{t\}$ such that $t' \in En(C,T)$ but $t' \notin En(C, T \cup \{t\})$.

  Consider the different reasons of why t' has become disabled when adding t to T.

  1. $source(t') \not\subseteq up(S)$, but C has not changed by adding t to T.
  2. t' is in structural conflict with some transition t'' in $T \cup \{t\}$. Since $T \subset En(C,T)$ we cannot have both t' and t'' in T. Hence one of them must be the newly added transition t. But then t would not have been enabled on (C,T), contradicting $t \in En(C,T) - T$.
  3. t' has the label e/a and while $m[\![e]\!](C,T) = \mathbf{T}$, $m[\![e]\!](C, T \cup \{t\}) \neq \mathbf{T}$. Since $t' \in T_0 = En(C, T_0)$, we have $m[\![e]\!](C, T_0) = \mathbf{T}$, where $T \cup \{t\} \subseteq T_0$. This contradicts the concavity property of event expressions, when applied to the sets $T \subseteq T \cup \{t\} \subseteq T_0$.

  It follows that the procedure cannot fail.

- The procedure cannot stop at $T \subset T_0$.
  The procedure can stop at $T \subset T_0$, only if $En(C,T) \cap (T_0 - T) = \phi$, contradicting the fact that T_0 is inseparable.

Let T be a set obtained by the construction. We show that T is an inseparable solution. Let the transitions in T be ordered in a sequence $t_1, t_2, \ldots, t_n$, according to the order of their addition to T. Clearly, since the construction stopped at T, we have that T satisfies

$$T = En(C,T).$$

It only remains to show that T is inseparable.

Consider any $T' \subset T$. Let t_k be the first transition, in the above ordering, which belongs to $T - T'$. We claim that $t_k \in En(C,T')$ which leads to the fact that $En(C,T') \cap (T - T') \neq \phi$. Assume to the contrary, that $t_k \notin En(C,T')$. Then we have three sets

$$T_1 = \{t_1, \ldots, t_{k-1}\}, \qquad T_2 = T', \qquad T_3 = T,$$

such that $T_1 \subseteq T_2 \subseteq T_3$, and yet

1. $t_k \in En(C, \{t_1, \ldots, t_{k-1}\})$
2. $t_k \notin En(C,T')$
3. $t_k \in En(C,T)$.

Claim 1 follows from the fact that the constructive algorithm picks transitions

to be added only if they are enabled under the current approximation. Claim 2 is our contrary assumption. Claim 3 follows from the fact that $t_k \in T = En(C,T)$.

We thus obtain a contradiction to the concavity property.

We must conclude that $En(C,T') \cap (T - T') \neq \phi$ for any $T' \subset T$, and hence T is inseparable.

7. Extending the Syntax

The syntax of event expressions, with its careful construction out of the subclasses of positive and negative event expression, was specially designed to guarantee the property of concavity. Let us consider several possible extensions of the syntax and show that they lead to non-concave behaviors.

Examples : Assume that $e, f, g \in \Pi$ are primitive event expressions, $X, Y, X', Y' \subseteq S$ are pairwise orthogonal sets and $v \in V$ is a variable.

- Allowing disjunction of a positive and negative event expression.
  The expression $e \vee \neg f \notin E$. passes through the path $(\mathbf{T}, \mathbf{F}, \mathbf{T})$ in the case that $T_1 = \phi$, $T_2 = \{(X, g/f!, Y)\}$ and $T_3 = \{(X, g/f!, Y), (X', g/e!, Y')\}$.
- Allowing conditions in negative event expressions.
  The expression $(\neg e)[new(v) = 0] \vee \neg f \notin E$ passes through the path $(\mathbf{T}, \perp, \mathbf{T})$ in the case that $T_1 = \phi$, $T_2 = \{(X, g/f!, Y)\}$ and $T_3 = \{(X, g/f!, Y), (X', g/v := 0, Y')\}$.
- Allowing conditions in positive event expressions.
  The expression $\neg(e[new(v) < 0]) \notin E$ passes through the path $(\mathbf{T}, \perp, \mathbf{T})$ in the case that $T_1 = \phi$, $T_2 = \{(X, g/e!, Y)\}$ and $T_3 = \{(X, g/e!, Y), (X', g/v := 0, Y')\}$.

7.1 Extensions by Transition Duplication

It is still possible to accommodate more general event expressions. Let us define first the most general syntax for event expressions. We define the class of *extended event expressions* $\mathcal{E}$ by

- Every event expression is an extended event expression. That is, $E \subseteq \mathcal{E}$.
- If e, e_1 and e_2 are extended event expressions, then so are $\neg e$, $e_1 \vee e_2$, $e_1 \wedge e_2$ and $e[c]$, where c is a boolean term.

It is straightforward to see that the evaluation function m also defines evaluation of extended event expressions over extended configurations $K = (C, T)$.

Lemma 5 : Every extended event expression is equivalent to a disjunction of event expressions.

Proof : Given an extended event expression, we can bring it to a disjunctive normal form. In performing this transformation we have to manipulate expressions involving events as well as boolean terms. Some of the rules for manipulating such combinations are given by the following equivalences

$$\begin{aligned}
\neg e[c] &= \neg e \vee \lambda[\neg c] \\
\lambda \wedge e &= e \wedge \lambda \ = \ e \\
(e_1 \wedge e_2)[c] &= e_1 \wedge (e_2[c]) \\
e_1[c_1] \wedge e_2[c_2] &= e_1 \wedge e_2[c_1 \wedge c_2] \\
(e_1 \vee e_2)[c] &= e_1[c] \vee e_2[c]
\end{aligned}$$

Using these and additional rules, we can bring every extended event expression to the form

$$E_1 \vee E_2 \vee \ldots \vee E_n,$$

Each disjunct E_i in this presentation is an event expression of the form

$$(e_1 \wedge \ldots \wedge e_m \wedge \neg f_1 \wedge \ldots \neg f_k)[c],$$

where $e_1, \ldots, e_m, f_1, \ldots, f_k$ are either primitive events or simple events of the form *entered*(s), *exited*(s), *assigned*(v).

Consider a statechart whose author wanted to have a transition t consisting of $(X, e/a, Y)$, where e is an extended event expression. Let e have the presentation $E_1 \vee E_2 \vee \ldots \vee E_n$ as a disjunction of event expressions. Then we suggest to the author of the original statechart to replace the single transition t by the n transitions

$$t_1 : (X, E_1/a, Y) \quad t_2 : (X, E_2/a, Y) \ \ldots \ t_n : (X, E_n/a, Y).$$

In this representation, the labels are event expressions and the overall effect is the same. In fact, the author may still prefer to present a single transition labeled by e, which is *interpreted* as the set of n transitions as shown above.

Another extension of the syntax we may allow is the reference to new values of variables on the right hand side of assignment actions. Such a reference, of the form $new(v)$, is allowed, provided the trigger of that transition includes the conjunct $assigned(v)$. This conjunct ensures that the transition will be taken only if the term $new(v)$ is defined. Thus the following is an admissible label

$$e \wedge assigned(y) \wedge assigned(z)/x := x + new\ (y) + new\ (z)$$

Another extension is the inclusion of ***conditional actions*** with boolean terms containing new subterms. These are actions of the form

$$if\ c\ then\ a_1\ else\ a_2,$$

where c is a boolean term that may contain subterms of the form $new(v)$ and a_1, a_2, are actions. The interpretation of such conditional actions is again obtained by transition splitting. Thus, we interpret the following transition

$$t : (X, e/a \cup \{\ if\ c\ then\ a_1\ else\ a_2\}, Y),$$

as though it was presented by the two transitions

$t_1 : (X, e[c]/a \cup \{a_1\}, Y)$, and
$t_2 : (X, e[\neg c]/a \cup \{a_2\}, Y)$.

This interpretation transforms any transition with conditional actions into a set of transitions, all of whose actions are unconditional.

The previous version of the syntax, as presented in [HPSS], allowed event expressions of the form $tr[c]$, $fs[c]$, which are considered to occur when the boolean term c changes from false to true, or true to false, respectively. These can be expressed in terms of the events $assigned(v)$ and references to $new\ (v)$. For example, the event $tr[x = y]$, can be expressed as the disjunction

$$\neg assigned(x) \wedge assigned(y)[(x \neq y) \wedge (x = new\ (y))] \vee$$
$$assigned(x) \wedge \neg assigned(y)[x \neq y) \wedge (\ new\ (x) = y)] \vee$$
$$assigned(x) \wedge assigned(y)[(x \neq y) \wedge (\ new\ (x) = new\ (y))]$$

Obviously, this expression is not a standard event expression but rather an extended expression. Its interpretation calls for splitting the transition into three copies, each labeled by one of the disjuncts.

Acknowledgements

We wish to thank Rivi Sherman for many constructive comments and identification of bugs in previous versions of the manuscript. We gratefully acknowledge many enjoyable discussions with W.P. de Roever, R. Gerth, C. Huizing,

J. Hooman, R. Koymans and R. Kuiper, which significantly contributed to our understanding of the tradeoffs between the various requirements expected from a good semantics. We thank Carol Weintraub and Sarah Fliegelmann for typing the numerous versions of the manuscript.

References

[BC] G. Berry, L. Cosserat – The Synchronous Programming Language ESTEREL and its Mathematical Semantics, Proceeding of CMU Seminar on Concurrency, Springer Verlag, *LCNS* **197** (1985), 389–449.

[BCH] J.-L. Bergerand, P. Caspi, N. Halbwachs – Outline of Real-Time Dataflow Language, Proceeding of IEEE-CS Real-Time Symposium, San Diego (1985).

[BG] G. Berry, G. Gonthier – The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation, Technical Report, Ecole Nationale Superieure des Mines de Paris (1988).

[H] D. Harel – Statecharts: A Visual Approach to Complex Systems, *Science of Computer Programming* **8** (1987) 231–274.

[HG] C. Huizing, R. Gerth – On the Semantics of Reactive Systems, Eindhoven University of Technology (1988).

[HGR] C. Huizing, R. Gerth, W.P. De Roever – Modelling Statecharts in a Fully Abstract Way, Colloquium on Algebras of Trees and Programs, Springer-Verlag *LCNS* **299** (1988).

[HPSS] D. Harel, A. Pnueli, J.P. Schmidt, R. Sherman – On the Formal Semantics of Statecharts, Symposium on Logic in Computer Science, (1987) 54–64.

[S] M. Shalev – On the Operational Semantics of Statecharts, M.Sc. thesis, Weizmann Institute (1988).

The Decent Philosophers: An exercise in operational semantics of concurrent systems

Wolfgang Reisig
Technical University of Munich
and GMD, Sankt Augustin

25 years of Jaco's extraordinarily successful work in semantics causes a particular occasion to arrange for him a big dinner with friends and colleagues. Edsger W. Dijkstra almost 20 years ago suggested how to organize such a dinner: The table is round, the dish is spaghetti, participants (called *philosophers*) share forks with their respective neighbours, using both their right and left fork when taking a meal. Each philosopher may take several meals during the dinner.

A lot of different courses of the dinner are possible in principle. As an example, a greedy philosopher might have several meals before his left neighbour had one, or a philosopher might starve to death due to conspiring neighbours.

The participants at Jaco's dinner of course all will be very decent: Each philosopher takes his left and his right fork alternating with his corresponding neighbour. In the following we study dinners of decent philosophers. It will turn out that this appears trivial only at a very first glance.

How to describe dinners of decent philosophers?

This question will be tackled depending on the number of participating philosophers.

In case two philosphers A and B, say, are the only participants, the decent dinner in the usual style is

$$\ldots a\; b\; a\; b\; a \ldots \tag{1}$$

Each occurrence of "a" and of "b" in (1) denotes a particular meal of A and of B, respectively. We are not interested in how the dinner begins or ends; so (1) describes the dinner in progress, without explicit bound to the left as well as to the right. (1) obviously indicates that the meals of the philosophers are taken sequentially.

In case three decent philosophers A, B and C participate in the dinner we likewise get the sequence

$$\ldots a\; b\; c\; a\; b\; c\; a \ldots \tag{2}$$

as well as the sequence

$$\ldots a\; c\; b\; a\; c\; b\; a \ldots, \tag{3}$$

the meaning of which is obvious from the above explanation of (1).

Four decent philosophers are enough to cause concurrency to some degree. Assuming the right neighbour of A, B, C and D to be B, C, D and A, respectively, we may represent their dinner as

$$\cdots \binom{a}{c} \binom{b}{d} \binom{a}{c} \binom{b}{d} \cdots \tag{4}$$

with $\binom{x}{y}$ denoting the philosophers X and Y to take a meal at the same time. Formally, (4) describes a sequence of sets of meals, a *step sequence*, whereas in (1), (2) and (3) we had sequences of single meals, i.e. *event sequences.*

Before continuing we should discuss which properties of the decent philosphers' dinner are represented in (1) - (4): The single meals of the involved philosphers are ordered exactly such that neighbouring philosophers' meals alternate. No further assumptions or restrictions are represented. Particularly no means are available to observe order of far apart philosophers meals. All order on the meals occurring through the entire dinner is imposed by the alternate ordering of neighbour's dinners.

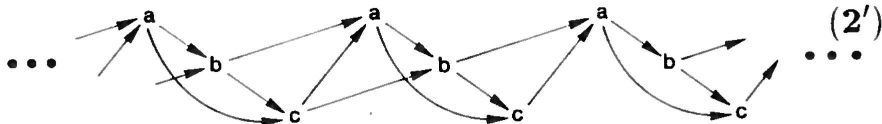
Now we turn to the case of five decent philosophers. One more or less quickly realizes that no event sequence and no step squence of meals would property describe their dinner: Each such sequence guaranteeing alternation among neighbours, inevitably causes unmotivated, additional order among far apart philosophers! So the quest is:

How to represent the dinner of five decent philosophers?

We tackle this problem by a fresh look at the representation of decent philosophers' dinner, taking now into account the forks, too. As a graphical convention, let $x \longrightarrow y$ denote a meal of the philosopher X, followed by the joint fork being passed over to a neighbour philosopher Y of X, followed by a meal of Y. This leads to a new representation of (2) as

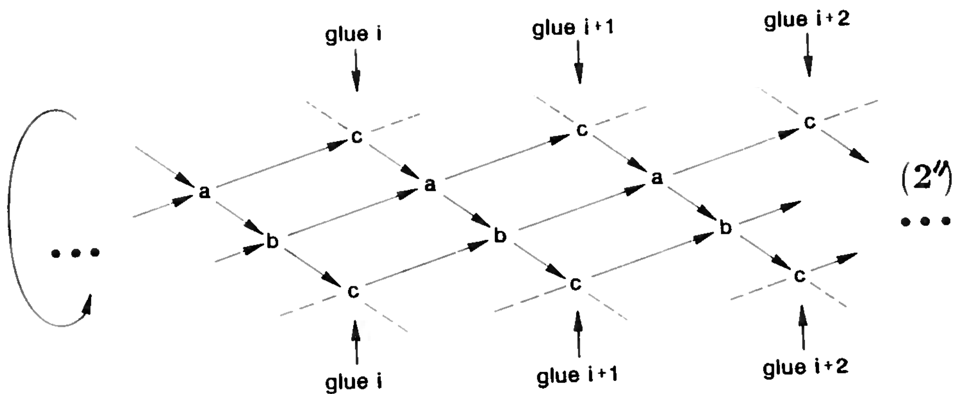
$(1')$

because the right as well as the left neighbour coincide in the two philosphers system. Consequently, for (2) we obtain (2′):

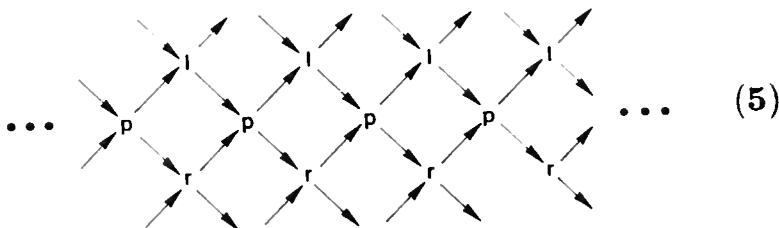


$(2')$

A three dimensional representation of (2′) shows its highly symmetrical structure: In the following figure (2″), unordered "c" are assumed to be identical, i.e. (2″) may be conceived an *inscribed cylinder* by glueing the unordered "c"–occurrences and also the straight and corresponding dotted arrows, as indicated.

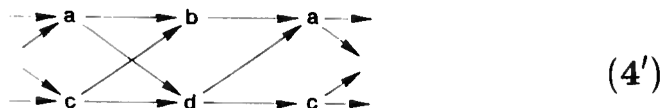


(2″) points at a general representation scheme of the decent philosophers' dinner: The meals of each philosopher P are represented by "p" occurrences along a virtual horizontal line, accompanied by $P's$ left an right neighbours meals "ℓ" and "r", according to the following scheme (5):

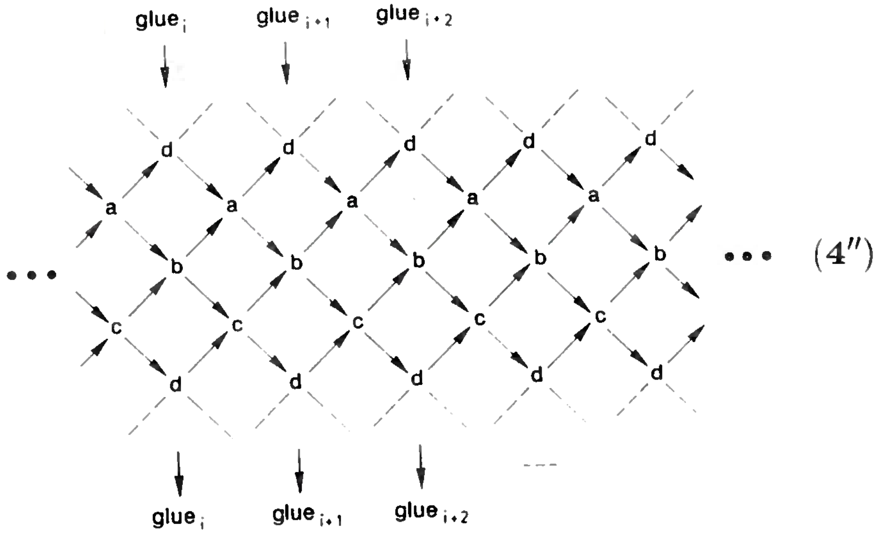


As an exercise we might ask for representations (3′) and (3″) corresponding to (3), just as (2′) and (2″) correspond to (2). (3″) in fact can be gained from (2″) by a different glueing policy: Glue the upper c in the "glueing cut" i with the lower c in cut $i + 1$. (This becomes even more obvious by a totally symmetrical representation as in (5)).

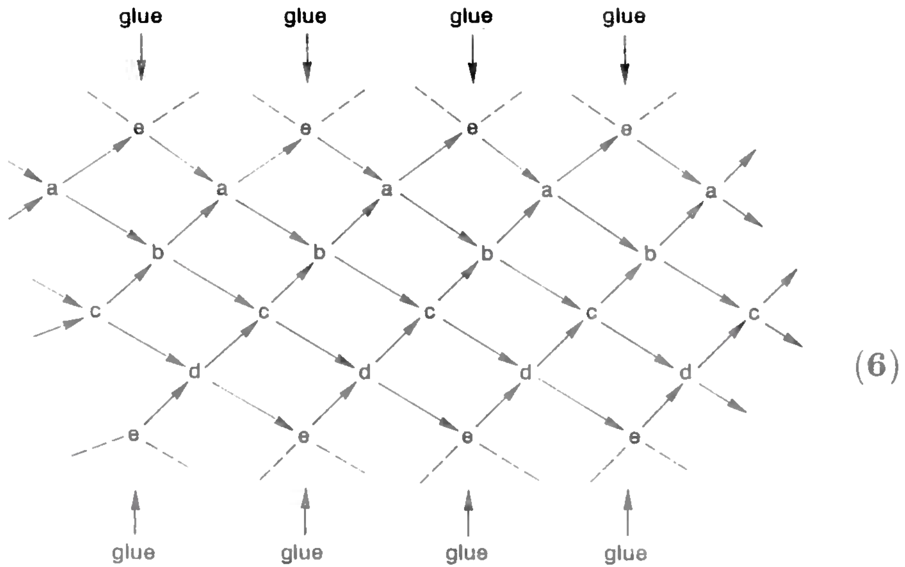
Turning to four philosophers, a revised representation of (4) is



which according to the scheme (5) has the following three-dimensional symmetrical representation (4″):



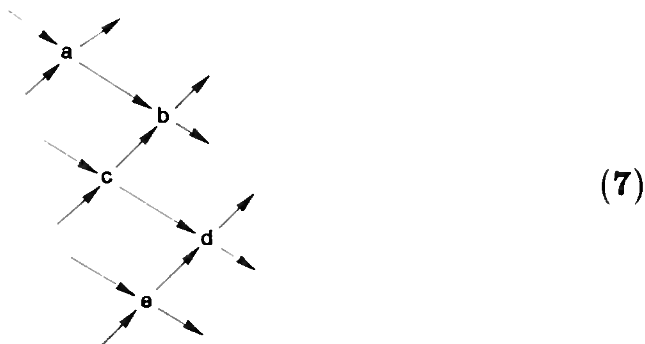
Based on the above considerations, particularly on (5), we have now the means to represent a dinner of five decent philosophers A, B, C, D, E:



What is now the essential structural difference between (4″) and (6)? Why has (4″) got a representation as a step sequence, (4), whereas (6) has not? Before discussing this topic we have to drop a word on what the figures (1′) ... (6) in this chapter formally denote. In fact they represent *labelled, partially ordered sets.* Their elements are visible only indirectly by their labels, the partial order is represented as usual in Hasse diagrams, with $x \longrightarrow y$ denoting an element labelled x, to be a direct predecessor of an element labelled y.

Now we observe that the partial order of (4″) (which was identical to that of (4′), has a *transitive complement*: This relation, representing the involved "unorder" (or "concurrency"), decomposes into two-elementary equivalence classes with elements labelled either "*a*" and "*c*" or "*b*" and "*d*", respectively.

(6) on the contrary has sub-structures such as shown in (7)



(7)

where "*e*" is unordered with "*a*" and "*a*" is unordered with "*d*", but "*e*" ordered with "*d*".

It generally holds that a dinner of decent philosophers can be represented as an event- or step-sequence only in case the concurrency involved (i.e. the complement of the meals' order) happens to be transitive.

What about the general case?

The consequences of the above considerations are far reaching: Operational semantics of non-sequential systems in general cannot be formulated in the framework of transition systems!

In order to overcome this problem in our concrete example, Jaco might avoid the dinner to have five participants. But any bigger number of participants will cause the same problem: (7) is a sub-structure of any dinner with five or more decent philosophers. So, one should instead base operational semantics of concurrent system onto the some what more general ground of partially ordered sets.

Some more or less interesting questions remain: Is there always a unique dinner for any number n of philosophers? Certainly not for $n = 3$, as we have seen above. But intuitively this case appears an exception, caused by the need for "too tight synchronization". How to formulate this formally? How to prove uniqeness of all dinners for $n \neq 3$?

More generally: What denotational semantics corresponds to partially ordered labelled sets? What should it abstract form? What reflexive domains are adequate? How get compositionality?

Jaco will surely find an adequate solution during another 25 years at CWI!

Programmacorrectheid en het college Inleiding Informatica

R.P. van de Riet

Faculteit Wiskunde en Informatica
Vrije Universiteit, Amsterdam

Februari 1989

1. Inleiding

Wanneer het verzoek bij je komt om een bijdrage te leveren voor het liber amicorum van Jaco de Bakker en je bent, zoals ik, studie- en jaargenoot van hem geweest, je hebt samen een tijd in de rekenafdeling van het toenmalige Mathematisch Centrum doorgebracht en je bent beide hoogleraar informatica aan dezelfde universiteit dan kun je dit verzoek niet weigeren.
Het onderwerp dat ik koos lag voor de hand: mijn ervaringen bij het geven van onderwijs, met name in het college Inleiding Informatica, op het gebied van programmacorrectheid. Het gaf me de mogelijkheid iets van de ervaringen van de afgelopen 15 jaar op een belangrijk gebied waar de invloed van De Bakker intensief is geweest te vertellen.
Dat ik daarbij de kans laat lopen over de ervaringen van de eerste tien jaren te vertellen neem ik maar voor lief; het bespaart me mijn ongenoegen te uiten over het feit dat, hoewel ik hem in vele functies vóór was (medewerker MC, sous-chef rekenafdeling, hoogleraar VU) hij het van mij gewonnen heeft met het vieren van dit vijf en twintig-jarig jubileum.
Ik kan hem daarmee van harte feliciteren en wil hem ook bedanken voor de vele jaren van voortreffelijke samenwerking die geweest zijn en die ongetwijfeld nog zullen komen.

De lezer vraag ik dit verhaal te lezen tegen de achtergrond van het gelegenheidsaspect ervan en rekening te houden met het feit dat ik geen deskundige ben op het gebied van de programmacorrectheid.

2. Het college Inleiding Informatica in de beginjaren

In 1970, toen ik bij de VU begon, startten Jim van Keulen en ik de cursus Inleiding Informatica dat uit een college bestond en een practicum. Voor het college werd een diktaat gebruikt dat ook bij het MC werd gebruikt voor het "Oriënterend Colloquium Informatica", een cursus voor leraren wiskunde. Het diktaat bevat vier hoofdstukken: "Inleiding", "algoritmen", "De werking van een Computer", waarin de Verschrikkelijk Eenvoudige RekenAutomaat (VERA een bijzondere uitvinding van mezelf; Wolbers gebruikte in die tijd dezelfde naam voor een Delftse uitvinding) en de MIX computer van Knuth [Knu] werden beschreven, en "De werking van een assembler geschreven in ALGOL 60". Het is duidelijk waar in die tijd mijn interesse naar uitging: programmeren en computers; correctheid kwam nog niet voor.

6 maart 1989

Het Koude Start Programma (KSP) van de MIX was van grotere interesse. Voor de liefhebbers: het bootstrapprogramma voor onze MIX implementatie (een ALGOL 60 programma van negen en een halve pagina's) las de inhoud van één kaart op een zekere plaats in het geheugen. Het KSP was bedoeld om gelezen te worden van die kaart zó dat de MIX computer daarna een willekeurig programma en data op veel meer kaarten kon lezen en verwerken. Curiositeitshalve volgen hier de 80 symbolen op die kaart:

0E02.0705A0100,0705U0D05A0100,0D05U001G=0015A0115H000050015U0100.0704¬0123456789

Het was een probleem bij het maken van de KSP dat een noodzakelijke instructie niet was voor te stellen door symbolen op de ponskaart, simpelweg omdat de interne code ervoor niet overeenkwam met een (MC-ALGOL-RESYM) code. De oplossing was de instructie te "bakken", d.w.z. eerst als data te laten uitrekenen. Dat deze oplossing vanuit het oogpunt van programmacorrectheid een erg slechte was wisten we toen niet. Hetzelfde idee werd (en wordt nog steeds) in LISP toegepast. In een latere versie van het collegediktaat Inleiding Informatica heb ik dat idee toegepast voor een computer die slechts één soort instructie kent, de "Een Instructie Computer" (EIC). Feitelijk werden in die computer vrijwel alle instructies eerst "gebakken". Er kwamen instructies in voor die "zichzelf" veranderen (een thema waarover ik het in de laatste paragraaf nog wil hebben). Uiteraard is het idee al van Turing afkomstig, die immers de Turing Machine voorziet van een tape waarop eerst data wordt gezet die daarna als instructies wordt gelezen.

De reden dat ik zo uitweid over die eerste cursus is om aan te geven waar toen bij mij de grootste interesse lag: problemen overwinnen die gesteld werden door bijzondere eigenschappen van de hardware. Ik kon me de luxe veroorloven die problemen in ALGOL 60 te bekijken, hoewel ik als verantwoordelijke voor het functioneren van de EL X8 computer ook exercities uitvoerde op het niveau van machine-instructies, en dat was bepaald veel lastiger.

Dat mijn aandacht momenteel veel meer gericht is op problemen die door de toepassingen worden gesteld, met name toepassingen op het gebied van informatiesystemen, is een tegenstelling die ik graag onderstreep. Het heeft ook veel te maken met een verschuiving van het **hoe** naar het **wat**, van imperatief naar declaratief denken, van programmeren naar specificeren. Dit laatste heeft dan weer nauw verband met programmacorrectheid.

Het eerste VU collegediktaat Inleiding Informatica is van 1972. Het gaat weer over algoritmen (als voorbeeld wordt een algoritme in mini-ALGOL gegeven die gebruikt kon worden om zich te begeven naar een protestdemonstratie, een actueel onderwerp in die tijd!), over de

programmeertaal ALGOL 60, representatie van informatie, booleans, integers en reals, machine code (VERA), talen en vertalen (waarin ik kon verwijzen naar eigen werk [vdR]), het operating systeem en de "Computer en Maatschappij" (een onderwerp dat nog steeds even actueel is als toen). Geen Programmacorrectheid.
In het diktaat van 1973 ontbreekt het hoofdstuk "Computer en Maatschappij", maar de eerlijkheid gebiedt te zeggen dat het eerder genoemde stuk slechts uit een halve pagina literatuurverwijzingen bestaat. Opmerkelijk is hier dat het werken met magnetische tapes wordt uitgelegd aan de hand van een programma om een straat in een stad te zoeken. Verder wordt uitvoerig ingegaan op compiler-compilers, inderdaad een interessant onderwerp, maar nog steeds geen programmacorrectheid.

3. Het college Inleiding Informatica in de midden periode

In 1973 werd De Bakker bij de VU benoemd om les te geven op het gebied van de Programmeertheorie. Blijkens het collegediktaat "Inleiding Programmeren" uit 1976 is de programmeertheorie de

"studie van theoretische grondslagen voor algoritmen".

Hier wordt de programmacorrectheid niet expliciet genoemd, maar inmiddels is de betekenis van dit begrip wel doorgedrongen in het collegediktaat Inleiding Informatica. In dat jaar verschijnt dit diktaat namelijk voor de eerste keer met een hoofdstuk over programmeermethoden, waarin ruime plaats aan het begrip programmacorrectheid wordt gegeven.
Het college krijgt dan een vorm die redelijk aan de intenties voldoet, namelijk een overzicht van het gehele gebied van de informatica zodat studenten bij het begin van de informaticastudie, toen nog ingebed in de wiskundestudie, een redelijk overzicht kregen van het geheel van de informatica. Zodat ook de docenten, en inmiddels waren er drie: Tanenbaum, de Bakker en ikzelf, hun colleges konden geven die enigszins herkenbaar pasten in een groter geheel. We achtten het van belang dat de studenten de samenhang met andere vakken reeds kenden alvorens een bepaald vak dieper gingen bestuderen. Aldus zouden ze ook beter gemotiveerd het betreffende college volgen. Met name vond en vind ik dit van belang voor de theoretische vakken, aangezien er een behoorlijk gevaar dreigt dat met name deze vakken door de studenten niet als echt relevant herkend worden en als "wiskundige ballast" gezien worden. Een feit is dat het tentamen Programmeertheorie door veel studenten wordt uitgesteld tot het laatste moment.

De manier waarop het onderwerp programmacorrectheid werd en nog steeds wordt behandeld is als volgt. Eerst wordt het Software Engineering probleem geïntroduceerd: dat het maken en onderhouden van software een dure aangelegenheid aan het worden is; dat dit met gestructureerd programmeren te maken heeft en met programmacorrectheid. Het sorteren van een array met reële getallen wordt als voorbeeld genomen. Het betreft een bepaald soort schuiven, zodat we het i.h.v. met "Schuifsort" zullen aanduiden. Er wordt een stroomschema (dit is de enige plaats in vermoedelijk de hele studie waar onze studenten het begrip stroomschema tegenkomen) opgesteld en aan de hand daarvan een Pascal programma (momenteel wordt Modula 2 gebruikt, want we gaan met de tijd mee!). Dit programma heeft een zeer ondoorzichtige structuur vanwege de goto's. (In Modula is er een complicatie omdat geen goto's gebruikt kunnen worden, zodat je ook niet kunt laten zien hoe slecht ze zijn). Dit wordt aangeduid als "ongestructureerd" programmeren. Daarna wordt hetzelfde programma opgezet met de bekende trits methoden: sequentie, conditioneel, repetitie en er wordt gedemonstreerd hoe veel beter leesbaar het resultaat is geworden: een "gestructureerd programma". De idee van successieve verfijning wordt vervolgens geïllustreerd. Daarna komt de programmacorrectheid aan bod.
Een natuurlijke vraag die gesteld en beantwoord wordt is: "je beweert dat Schuifsort dat array sorteert, maar kun je dat ook bewijzen?"

Voor mij, opgeleid als wiskundige, is dit altijd weer een boeiende vraag. Je moet uitleggen wat een bewijs is. In de eerste jaren deed ik dat niet goed. Later kwam John-Jules Meyer, medewerker van De Bakker, met uitbreidingen in de vorm van bewijsregels, correctheidsformules, enz. waardoor het diktaat aanzienlijk verbeterde. De eerste keer dat ik er over schreef meende ik te kunnen volstaan met de bekende Floyd axioma's, die ik nu als volgt noteer:

voor de sequentiële structuur:

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S\ ;T\ \{R\}}$$

voor de if-then-else structuur:

$$\frac{\{P\wedge B\}\ S\ \{Q\},\{P\wedge\neg B\}\ T\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S \text{ else } T\ \{Q\}}$$

6 maart 1989

voor de repetitiestructuur:

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \text{while } B \text{ do } S\ \{P \wedge \neg B\}}$$

Bovendien voerde ik als axioma voor de assignment statement in:
"als de uitspraak B(a) waar is vóór de assignment "b:= a" dat dan, in principe, de uitspraak P(b) waar is na de assignment". (Let op: "in principe") Met deze hulpmiddelen werden "bewijzen van correctheid" gegeven voor de delingsalgoritme, de algoritme van Euclides (uiteraard dat moet), twee sorteringsalgoritmes, Maxsort (gebaseerd op het vinden van maxima) en Schuifsort en het werd toegepast bij de behandeling van Quicksort. Bij Euclides gebruikte ik als definitie voor grootste gemene deler de bekende eisen van deelbaarheid. Als loop-invarianten nam ik er twee: een die aangeeft dat delers van de uitgangsgetallen n_0 en m_0 ook delers zijn van de variabelen n en m en een die aangeeft dat n_0 en m_0 steeds lineaire combinaties zijn van n en m. Het simpele feit dat aan het eind van de algoritme de rest bij de laatste deling van m door n nul oplevert geeft het bewijs dat de gevonden n ook een deler is van n_0 en m_0. Helaas is na verhitte discussies dit, in mijn ogen fraaie, bewijs vervangen door het bekende bewijs dat gebaseerd is op de stelling: ggd(m,n)= ggd(n, m mod n). Dat mijn bewijs gebaseerd was op de primaire eigenschappen van de ggd woog in de ogen van anderen helaas niet op tegen de veel eenvoudiger loop invariant.
Belangrijker dan dit detail was echter dat ik aan drie gevallen, waarvan de complexiteit niet al te simpel was, namelijk Maxsort, Schuifsort en Quicksort, kon laten zien dat het werken met correctheidsformules in de praktijk te doen was. Het is in dit verband relevant te vermelden dat Schuifsort een programma is dat uit een stuk of tien korte regeltjes bestaat en dat het "bewijs van correctheid" vier (ruim) gedrukte pagina's beslaat. We laten een gedeelte van die tekst als illustratie zien:

"

Het schuiven hebben we in fig. 5.3 in beeld gebracht:

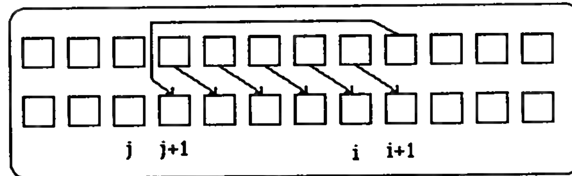


Fig. 5.3. Het schuiven in *schuifsort*.

Na afloop van het schuiven moet het volgende waar zijn (dat echter nog wel bewezen moet worden):

$$
\begin{aligned}
S\colon \exists j\colon\ & 0 \le j < \mathbf{i} \\
& \wedge (\forall k{:}\, j+1 \le k \le \mathbf{i} \Rightarrow A_1[k] = \mathbf{A}[k+1]) \\
& \wedge (\forall k{:}\, 1 \le k \le j \vee \mathbf{i}+2 \le k \le \mathbf{n} \Rightarrow A_1[k] = \mathbf{A}[k]) \\
& \wedge (\mathbf{A}[j+1] = \mathbf{Ai} \wedge \mathbf{A}[j+2] > \mathbf{Ai}) \wedge (j > 0 \Rightarrow \mathbf{A}[j] \le \mathbf{Ai})\ .
\end{aligned}
$$

[..]

Rest ons nu nog de uitspraak S te bewijzen die na de algoritme voor *schuif* moet gelden. Deze algoritme luidt:

schuif:

```
11:  j:=i-1;
     while (j>0) and (A[j]>Ai) do
       12:  A[j+1]:=A[j];
       13:  j:=j-1
     end;
14:  A[j+1]:=Ai;
15:
```

Te bewijzen is dat als vooraf geldt:

$$
\begin{aligned}
& (\forall k{:}\ 1 \le k \le \mathbf{n} \wedge k \ne \mathbf{i}+1 \Rightarrow A_1[k] = \mathbf{A}[k]) \\
& \wedge\, perm(A_0, A_1) \\
& \wedge\, A_1[\mathbf{i}] > A_1[\mathbf{i}+1] \\
& \wedge\, \mathbf{Ai} = A_1[\mathbf{i}+1] \\
& \wedge\, \mathbf{A}[\mathbf{i}+1] = A_1[\mathbf{i}]
\end{aligned}
$$

dat dan na de schuifoperatie de uitspraak S geldt.
Als loop-invariant kiezen we een iets gewijzigde S:

Q:

$$\forall k\text{: } \mathbf{j}+1 \le k \le \mathbf{i} \Rightarrow A_1[k] = \mathbf{A}[k+1] \qquad (1)$$

$$\wedge\ \forall k\text{: } (1 \le k \le \mathbf{j} \vee \mathbf{i}+2 \le k \le \mathbf{n}) \Rightarrow A_1[k] = \mathbf{A}[k] \qquad (2)$$

$$\wedge\ \mathbf{A}[\mathbf{j}+2] > \mathbf{Ai}\ . \qquad (3)$$

Initiëel geldt met $\mathbf{j}=\mathbf{i}-\mathbf{1}$:

$$A_1[\mathbf{i}] = \mathbf{A}[\mathbf{i}+1],$$

dus (1) geldt;

$$\forall k\text{: } (1 \le k \le \mathbf{n} \wedge k \ne \mathbf{i}+1) \Rightarrow A_1[k] = \mathbf{A}[k],$$

dus ook (2) geldt;

$$\mathbf{A}[\mathbf{i}+1] = A_1[\mathbf{i}]\ > A_1[\mathbf{i}+1] = \mathbf{Ai},$$

dus (3) geldt ook.

Als we de invariantie van Q bewezen heben, dan volgt uit het niet voldoen aan de conditie in het while-statement en uit $\mathbf{A}[\mathbf{j}+\mathbf{1}]{:=}\mathbf{Ai}$ de gevraagde bewering S, immers: $\mathbf{A}[\mathbf{j}+\mathbf{1}]=\mathbf{Ai}$ volgt uit het zojuist genoemde assignment statement, $\mathbf{j}>0 \Rightarrow \mathbf{A}[\mathbf{j}] \le \mathbf{Ai}$ volgt uit het niet voldoen aan de conditie en het bestaan van $\mathbf{j}$ met $0 \le \mathbf{j} < \mathbf{i}$ is ook verzekerd.

Tenslotte tonen we nu de invariantie van Q aan:
Aangenomen dat Q geldt bij 12. Bij 12 geldt bovendien:

$$\mathbf{j}>0 \wedge \mathbf{A}[\mathbf{j}] > \mathbf{Ai}\ .$$

Voor $k=\mathbf{j}$ geldt op grond van (2): $A_1[\mathbf{j}] = \mathbf{A}[\mathbf{j}]$. Na de assignment $\mathbf{A}[\mathbf{j}+\mathbf{1}]{:=}\mathbf{A}[\mathbf{j}]$ geldt dus in 13: $A_1[\mathbf{j}]=\mathbf{A}[\mathbf{j}+\mathbf{1}]$, zodat

$$\forall k\text{: } (\mathbf{j} \le k \le \mathbf{i} \Rightarrow A_1[k] = \mathbf{A}[k+1] \wedge (1 \le k \le \mathbf{j}\text{-}1 \vee \mathbf{i}+2 \le k \le \mathbf{n}) \Rightarrow A_1[k] = \mathbf{A}[k]\ ).$$

Uit $\mathbf{A}[\mathbf{j}] > \mathbf{Ai}$ volgt $\mathbf{A}[\mathbf{j}+1] > \mathbf{Ai}$.
Maar dit betekent, na substitutie van $\mathbf{j}$-1 door $\mathbf{j}$, dat we Q weer terug hebben, waarmee de invariantie van Q is aangetoond.
"

Met opzet is dit stuk van het diktaat gekozen omdat het een aantal zaken goed laat zien:

• de formules zijn door hun veelheid complex;

• de redeneringen worden opgehangen aan plaatsen in de tekst van het programma: als een of andere uitspraak bij 12 geldt dan volgt, na enige redenaties, dat een andere uitspraak bij 13 geldt;

• het gebruik van verschillende fonts is niet geheel consistent: moet je nu j+1 of j+1 schrijven in een logische uitdrukking, waar j een variabele is die in het programma voorkomt; de tekst van een programma wordt met *schuif* aangeduid;

• het bewijs van correctheid loopt mee met de volgorde van de tekst: van links naar rechts. Dit geldt ook voor assignment statements, zoals blijkt uit de laatste aangehaalde zin.

Hiermee kom ik op een moeilijk punt: mijn gebruik van het assignment statement axioma was niet correct maar het werkte prima in de bewijsvoering.

4. Het college Inleiding Informatica in de laatste jaren

In het collegediktaat van 1984 wordt het assignment statement als volgt ingevoerd:

Als men van een uitspraak R wil bewijzen dat deze geldt ná het assignment statement:

v:= expr

dan is het voldoende als aangetoond wordt dat de uitspraak R[expr/v] geldt vóór dat statement. Dit is de regel:

{P[expr/v]} v:= expr {P}

die ook in het bekende boek van Alagi´c en Arbib [A&A] wordt gebruikt.

In dit collegediktaat worden de axioma's op wat meer formelere wijze ingevoerd, waarin de hand van eerder genoemde John-Jules is te zien. Eenvoudiger voorbeelden worden nu behandeld om de begrippen te verduidelijken. Alles van rechts naar links in verband met de werking van het assignment statement. Voor die eenvoudige voorbeelden werkt dit ook prima, zo wordt

$\{x = x_0 \wedge y = y_0\}$

h:=x; x:=y; y:=h

$\{x = y_0 \wedge y = x_0\}$

6 maart 1989

recht toe recht aan bewezen door uit te gaan van de laatste uitspraak. In het geval van een programma dat het maximum berekent van een array wordt de methode ook toegepast, maar het resultaat van de redenering van twee pagina's is toch weer zo complex dat er behoefte is het als volgt samen te vatten:

"

```
m:=1; i:=2;
{ m=1 ∧ i=2 }
while i ≠ n+1 do
      { P: 1≤m<i ∧ ∀j:1≤j<i ⇒ A[m]≥A[j] }
      if A[m]<A[i] then
            { i≥1 ∧ ∀j:1≤j≤i ⇒ A[i]≥A[j] }
            m:=i
      else ε end;
      { 1≤m≤i ∧ ∀j:1≤j≤i ⇒ A[m]≥A[j] }
      i:=i+1
      { P: 1≤m<i ∧ ∀j:1≤j<i ⇒ A[m]≥A[j] }
end
{ 1≤m≤n ∧ ∀j:1≤j≤n ⇒ A[m]≥A[j] } .
```

"

Het probleem met die samenvatting is dat het geen echt bewijs is. Je kunt ook niet meer zien of je van links naar rechts gegaan bent of omgekeerd. Toch vragen we van onze studenten op het tentamen een bewijs à la deze samenvatting te geven want het verhaal dat er bij hoort is te lang. Overigens blijkt het onderwerp correctheidsbewijs voor de student het moeilijkste onderdeel van het tentamen te zijn.

Op mijn verzoek aan de "theoretische groep" de bewijzen van correctheid voor Maxsort, Schuifsort en Quicksort te veranderen zó dat ze wel van rechts naar links lopen is, begrijpelijkerwijs, tot nu toe niet ingegaan. Zelf zie ik de noodzaak om de vier pagina's tekst om te werken tot een veelvoud daarvan ook niet zo erg zitten.

Nu zou het probleem opgelost worden als ik naast het bovengegeven axioma voor de assignment statement de volgende axioma kies, uit het bekende boek van de Bakker [deB]: (p. 38)

"$\{P\}$ x:= s $\{ \exists y[P[y/x] \wedge x = s[y/x]]\}$,

where y is some variable such that $y \neq x$, and $y \notin ivar(P,s)$"

6 maart 1989

Dit axioma werkt namelijk van links naar rechts. Het zal echter een ieder duidelijk zijn dat je met zo'n axioma niet moet komen bij eerste jaars informaticastudenten.
Hier blijft dus sprake van een dilemma.

5. Evaluatie

Zoals uit het voorgaande blijkt is de laatste jaren in de vakgroep met vereende inspanning gewerkt aan een goede introductie van het onderwerp programmacorrectheid. Helaas zijn we er nog niet in geslaagd de studenten een zodanige vorm van werken met correctheidsformules te presenteren die ook geschikt is voor grotere programma's. Zelfs de korte sorteerprogramma's vergen erg veel formules, en eerlijk gezegd kon ik het ook niet opbrengen alle details van die bewijzen aan de studenten voor te zetten. Na de behandeling van Maxsort doorgeworsteld te hebben maak ik me er in het college vanaf door te zeggen dat het bewijs van correctheid van Schuifsort op soortgelijke wijze gaat, zij het dat de formules nog wat langer zijn: "zie diktaat".

Een veel ernstiger probleem dook echter op: de gebruikte axioma's voor de assignment statement zijn niet goed. Ze gelden alleen als je eenvoudige variabelen hebt, maar ze gelden niet meer als je met arrayelementen werkt en al mijn "echte" voorbeelden betreffen sorteeralgoritmen werkend op arrays! De Bakker vertelde me van zijn vondst; het was 1980 of daaromtrent. Ik was zo onder de indruk dat ik een week later collega van Oorschot, die toen nog Directeur bij de PTT was, er over vertelde. Vanaf die tijd staat het fenomeen ook in het collegediktaat. Het betreffende stuk tekst luidt als volgt:

"

Overigens is een waarschuwend woord hier op zijn plaats: de gebruikte regels (uitgevonden door Floyd en verbeterd door Hoare) werken alleen goed voor eenvoudige "normale" gevallen. Het was De Bakker die aantoonde dat het axioma voor het assignment statement niet voldoende is om adequaat voor alle mogelijke ingewikkelde assignment statements de betekenis vast te leggen. Beschouw de volgende schijnbaar juiste correctheidsformule:

$$\{ A[1]=1 \wedge A[2]=1 \}$$
$$A[A[1]]:=2;$$
$$\{ A[A[1]]=2 \}$$

Inderdaad, als we in de uitspraak $A[A[1]]=2$ de uitdrukking $A[A[1]]$ letterlijk substitueren door 2 dan ontstaat $2=2$, hetgeen waar is. Echter als $A[1] = A[2] = 1$ bij het begin van het

assignment statement dan is de waarde van A[A[1]] na afloop gelijk aan 1, hetgeen in tegenspraak is met bovenstaande correctheidsformule!
Omdat we van bewijsregels verwachten dat ze *altijd* juist zijn en niet alleen maar *meestal*, (we wilden immers weten dat ons programma *altijd* werkt) moeten we nog één extra afspraak maken:

> In correctheidsformules gebruiken we arrays alleen in de vorm:
>
> A[u]
>
> waarin u een uitdrukking is waarin gehele getallen en integer variabelen voor mogen komen, maar *geen* arrays.

In zijn boek [deB] geeft De Bakker een ander, meer ingewikkeld, axioma voor het assignment statement. Daarmee kunnen correctheidsformules bewezen worden waarin een array-index een willekeurige expressie mag zijn. We gaan er hier (uiteraard) niet verder op in.
"

Inderdaad heeft de Bakker in dit boek een vijftiental pagina's nodig om de subscripted variabelen te introduceren en de betekenis van assignment in de vorm van van vier definities en drie stellingen. Het is uitgesloten zulke zaken te behandelen voor eerste jaars.

Voor mij, als liefhebber van Hofstadter's "Gödel, Escher, Bach"[Hof], zat er in de statement

A[A[1]]:=2

wel een mooie aanleiding het een en ander over zelfreferentie te zeggen. Immers hier hebben we het geval van een statement die zijn eigen betekenis wijzigt (ik weet niet of je mag zeggen "die zichzelf wijzigt"): was de betekenis eerst:

A[1]:=2

na de uitvoering ervan is de betekenis gewijzigd in:

A[2]:=2

Het gaf me de mogelijkheid om het een en ander te zeggen over programma's die zichzelf als input hebben (denk aan compilers, maar ook aan Turing's halting probleem en aan Gödel's onvolledigheidsstelling), over recursieve procedures, over vorm en inhoud en de verschillende niveaus van machinearchitectuur, over Bach die zijn eigen naam als thema gebruikt, over Escher's zelfportretten, kortom een bonte lijst van interessante zaken.
Instructies die, zoals in het "Koude Start Programma" van de eerste versie van het collegediktaat, eerst "gebakken" moeten worden, zijn verdwenen uit het diktaat; daarvoor in de plaats zijn statements gekomen die zichzelf wijzigen, wat natuurlijk veel interessanter is.

Nog enkele opmerkingen over het collegediktaat Inleiding Informatica: Het hoofdstuk over

machinearchitectuur is nu zo gewijzigd dat het begint met de elementen van eerste orde logica en in no time overgaat naar Boole'se Algebra, sequentiële circuits, poorten en de electronica van een chip, inclusief optellers en geheugen; dit alles als prelude op het boek van Tanenbaum [Tan]. Overigens, was er een goed boek van Pohl en Shaw [P&S] dat we een groot aantal jaren gebruikten met name voor dit onderwerp. Het hoofdstuk "Computers in de Samenleving" beslaat nu drie pagina's, dat is niet overdreven veel. Als compensatie merk ik op dat de laatste twee uren van het college normaliter aan dit onderwerp besteed worden met allerlei gadgets, dia's, audio (uit de laatste fuga van de Kunst der Fugen waarin Bach naar zichzelf refereert) en de laatste tijd ook video (over robots enzo).

Tenslotte, we zien in de laatste jaren een veel grotere interesse van informatici voor specificaties. De reden is dat de systemen zo complex worden dat er steeds hoger niveau geprogrammeerd moet worden met machtige hulpmiddelen als grafische tools (denk aan op plaatjes gebaseerde Software Engineering tools) en logische programmeertalen als Prolog (zie [C&M]).

De bedoeling van Prolog was een taal te zijn waar de specificatie samengaat met het programma: het programma is de specificatie. Helemaal gelukt is dat niet: je moet nog teveel imperatief programmeren, nog te veel het hoe aangeven. Toch is het een stap in de goede richting gezet.

Een goed begrip van deze specificatietalen en het belang ervan krijg je door een goed inzicht te hebben in de achtergronden van programmacorrectheid. Daarom denk ik dat het onderwerp nog lang in het diktaat Inleiding Informatica opgenomen zal blijven, ondanks de gesignaleerde problemen.

Literatuur

[A&A] S. Alagi´c, M.A. Arbib: The design of Well Structured and Correct Programs, Springer, 1978.

[deB] J.W. de Bakker: Mathematical Theory of Program Correctness, Prentice Hall 1980.

[C&M] W.F. Clocksin, C.S. Mellish: Programming in Prolog, Springer, 1881.

[Hof] D.R. Hofstadter: Gödel, Escher, Bach: an Eternal Golden Braid, Basic Books, New York 1979.

[Knu] D.E. Knuth: The Art of Computer Programming Vol.3: Sorting and Searching,

Addison-Wesley, 1973.

[P&S] Ira Pohl, Alan Shaw: The Nature of Computation: an Introduction to Computer Science, Computer Science Press, 1981, ISBN 0-914894-12-9.

[vdR] R.P. van de Riet: ABC ALGOL, a portable language for formula manipulation systems, part 1 and part 2, Mathematical Centre tracts 46 en 47, Mathematisch Centrum, Amsterdam, 1973.

[Tan] A.S. Tanenbaum: Structured Computer Organization, Second Edition, Prentice Hall 1984.

APPLICATIONS OF COMPUTABILITY THEORY OVER ABSTRACT DATA TYPES

J V TUCKER

Centre for Theoretical Computer Science
University of Leeds
Leeds LS2 9JT
*England**

To J W de Bakker.

PROLOGUE

Prouder cities rise through the haze of time,
Yet, unenvious, all men have found is here.
Here is the loitering marvel
Feeding artists with all they know.
Vernon Watkins

Amsterdam is one of the important cities of the world for computer science. It is the most important city for my own interests in the mathematical theory of programming and programming languages. It is a place where many people struggle with research of lasting value. It is always a pleasure for me to be welcomed here, and especially to celebrate Jaco de Bakker's jubilee at the CWI. Jaco is one of the architects of this city's outstanding reputation in computer science, through his personal research, encouragement of the subject and colleagues, and his judgement and belief in high standards. One cannot do better than be a worthy example to others.

I will lecture on applications of computability theory over abstract data types. I will show how this subject has significant applications in widely different areas of computer science. This is due to the fundamental nature of the concept of an abstract data type, as modelled by class of abstract algebras. It was in Amsterdam that I began to realise the beauty, and utility, of the combination of computability and algebra in computer science.

As a mathematics student at Warwick and Bristol, I was interested in computability and its applications, principally to algebra, influenced by J P Cleave and J C Shepherdson. But as a research fellow at Oslo (reading the work of E Engeler, Z Manna and others, for example) it became clear to me that to make progress with the subjects that interested me, I had to master the exciting mathematical concepts and results arising in theoretical computer science. In that period I was also encouraged by disussions with J A Bergstra, on his visits to our group of generalised recursion theorists in Oslo. He was one of the Logicians of Utrecht who had recently turned to theoretical computer science.

In January 1979, I arrived in Amsterdam excited at the opportunity of learning about the theory of programming languages, and of developing and applying my knowledge of mathematical logic in the context of computer science. The opportunity to learn was arranged by Jaco de Bakker. I wrote to him about my work, asking for assistance and advice about moving into computer science; we had not met. A mathematical logician interested in computer science was welcome in very few places in the world, but Jaco's department at the Mathematisch Centrum was one: I joined J I

* *From August 1989*: Department of Mathematics and Computer Science, University College of Swansea, Singleton Park, Swansea, SA2 8PP, Wales.

Zucker; K R Apt had recently left for Rotterdam; J A Bergstra and J W Klop joined later. And to complete the list of those who influenced me, I must mention the computer scientist R J R Back, who arrived with logical interests arising from his theory of program refinement.

I brought with me to Amsterdam one book: Fuchs' *Dutch art.* I soon acquired another: the typescript of Jaco de Bakker's *Mathematical theory of program correctness.* I studied this with help from Jeff Zucker, with whom I shared an office. Clearly all that was missing from computer science were the algebras...

1. INTRODUCTION

In the monograph Tucker and Zucker[88], the semantic and correctness theories of **while** programs, **while array** programs, and recursive programs established in de Bakker[80] are generalised to allow programs that compute not simply on the natural numbers but on any abstract data type. In addition, error or exceptional states are introduced into the semantics, which arise when a variable is called in a computation without first being initialised. These enhancements involved mathematical work on:

many sorted algebras A and classes K of many sorted algebras;
error and exceptions in semantics and proof rules;
weak second order logic assertion languages;
theory of computable functions on A and K.

But the basic semantical theory of programs we relied on is that in de Bakker[80]. The ideas and results involved in the two books have proved to be a significant influence on our subsequent research and teaching. We began planning our book in 1979 and owe a great debt to Jaco de Bakker for his unfailing support over those years of composition.

In this lecture I will sketch some of the subjects I am currently studying, using methods that are dependent on those in Tucker and Zucker [88], particularly on the computability theory for classes of many sorted algebras. The subjects are:

synchronous concurrent algorithms and their application in hardware design; and
logic programming modules and abstract data types.

These apparently disparate applications are part of a programme, based on abstract data type theory, that aims at the formulation and analysis of the many interesting notions of *computability*, *specifiability*, and *verifiability* that exist in differents areas of computer science.

In Section 2 some important concepts concerning abstract data types will be explained, including the technical role of classes of algebras.

Section 3 summarises the computability theory on abstract data types as it stands in Tucker and Zucker[88]; it includes the *Generalised Church-Turing Thesis* for the process of *deterministic computation* on abstract data types. This computability theory is founded upon recursive functions on algebras and classes of algebras. An important point is that it distinguishes between *primitive recursion* and *course-of-values recursion,* as these are no longer equivalent on absract algebras. The computability theory was developed to prove that important sets of program states, such as the *weakest preconditions* and *strongest post conditions* of programs and assertions, were expressible in a weak second order many sorted logical language. (At the heart of this exercise is a theorem that represents generalised recursively enumerable sets in the language.)

In Section 4 the idea of a synchronous concurrent algorithm (sca) is defined. This is a network of processors and channels that compute and communicate in parallel, synchronised by a global clock. Such algorithms compute on infinite streams of data and are characteristic of hardware. This type of computation is formalised using the course-of-values recursive functions over classes of stream algebras. The study of this and other models of these clocked algorithms, and their application, is a substantial task: it aims at a general mathematical theory of computation based on hardware. The special case of unit delays throughout the network corresponds with the use of primitive recursive functions and has been studied intensively in joint work with B C Thompson (Manchester). Significant contributions to the theory, case stuides and software have been made by: K Meinke (Manchester), N A Harman (Swansea); S M Eker and K Hobley (Leeds); and A R Martin (Infospec Computers Ltd).

In Section 5 the idea of a module appropriate for logic programming is examined. This leads

to the need to distinguish between specification and computation, and to begin work on the scope and limits methods for the specification of relations on abstract data types, to complement the computation theory described in Section 3. This work with J I Zucker is a natural continuation of our studies of subjects we met in the preparation of our book.

I should also remark here that the study of some the semantic issues raised in Tucker and Zucker[88], whose roots lie in de Bakker[80], has also been continued in C Jervis [88].

2. ABSTRACT DATA TYPES

2.1 Modules and Algebras

In the theory of abstract data types, computation is characterised by a *module* whose semantics is a class K of many-sorted *algebraic structures* or *models*. A many-sorted algebraic structure or model A consists of a number of sets of data, and operations, and relations on the data. Such a structure, possibly satisfying some further properties, can be used to model semantically a *concrete implementation* of a module. A class K of structures, again possibly satisfying some further properties, can be used to model semantically the module abstractly as a *class of implementations*. For example, one standard definition of an abstract data type is that it is an isomorphism type i.e. a class of all isomorphic copies of an algebra: see Goguen, Thatcher, Wagner and Wright [78]. Among the properties of algebras that we will use to characterise meaningful or useful implementations are: *minimality, initiality, and computability* (see 2.3).

See Ehrig and Mahr [85] and Meinke and Tucker [89] for background material on algebra.

2.2 Modularisation

Consider the general idea that programming can be formulated as the creation of modules:

Programming creates a program module P from a component module C in order to obtain a task module T.

Consider a programming task in the case when the modules define implementation algebras. Suppose we want to compute a function F and a relation R on a set A using operations f_i and relations r_i. Then the *task algebra* is (A: F,R) and the *component algebra* is (A: f_i : r_i). The program may introduce a new data set B and functions g_j and relations s_j leading to *program algebra* (A, B: f_i, g_j, F: r_i, s_j, R). Of course this generalises to the case of many sets in the obvious way. To a user the finished product is the task algebra, and only the function F and the relation R are visible for the f_i, g_j and r_i, s_j are hidden. When, in the normal case, the modules define *classes* of implementation algebras we uniformise the above. These ideas are derived from those of R Burstall and J Goguen.

2.3 Classes of models

For the purposes of specification and verification, we expect K to be a subclass of Mod(T), the class of all models of some axiomatic theory T. Among the classes of interest are:

K=Mod(T)	K={A ε Mod(T): A is finite}
K= {A ε Mod(T): A is computable}	K= {A ε Mod(T): A is semicomputable}
K={A ε Mod(T): A is initial}	K={A ε Mod(T): A is final}

Each of these classes formalises an interesting aspect of the semantics of T. For example, consider initiality and computability. If model A is initial then it is (isomorphic to) a certain form of *standard implementation* by computer. To study the computation of F and R with respect to this K is to study the properties of a program module for F and R uniformly across all standard implementations.That a model A is computable means that it is *implementable in some way* by computer, according to the classical Church-Turing Thesis. To study the properties of F and R with respect to this K is to study the properties of a program module for F and R uniformly across all possible computer implementations.

2.5 Higher-order computation and logic

Many sorted algebra and logic may be employed as a unified framework for representing higher order computation and logic. For example, two forms of second order computation and logic over

a structure A can be represented by applying first order computation and logic to the extended structures, $\underline{A}$ and A^*, being A with streams, and arrays adjoined (with appropriate evaluation operations), respectively.

2.5.1 Augmentation of time cycles and streams Let A be an algebra and add to the carriers of A a set T of natural numbers and, for each carrier A_i of A, the set $[T \to A_i]$ of functions $T \to A_i$ or *streams* over A_i. To the operations of A we add the simple constant 0 and operation of successor t+1 on T, and

$eval_i : T \to A_i$ defined by $eval_i(t,a) = a(t)$.

Let this new algebra be $\underline{A}$.

2.5.2 Augmentation of arrays Let A be an algebra and add to the carriers of A a set of natural numbers T and the simple constant 0 and operation of successor t+1 on T. Next add an undefined symbol u_i to each carrier A_i of A, and extend the operations by strictness. This forms an algebra A_u with carriers $A_{i,u} = A_i \cup \{u\}$. We model a *finite array* over A_i by a pair (α, l) where

$\alpha: T \to A_{i,u}$ and $l \in T$ such that $\alpha(t) = u$ for all $t > l$.

Thus the pair is an infinite array, uninitialised almost everywhere. Let A_i^* be the set of all such pairs. We add these sets to the algebra A_u to create carrirers of the array algebra A^*. The new constants and operations of A^* are: the everywhere undefined array; and functions that evaluate an array at a number address; update an array by an element at a number address; evaluate the length and update the length.

We will use both extensions of A in the next sections.The second augmentation is an enhancement of the array algebras in Tucker and Zucker [88] that we made in Tucker and Zucker [89]. There are many interesting extensions and potential applications.

3. COMPUTABILITY THEORY

3.1 Computable Functions on Abstract Data Types

In Tucker and Zucker[88] we have examined some classes of functions over an adt that are generated from its basic operations by means of

sequential composition;
parallel composition;
simultaneous primitive recursion;
simultaneous course of values (cov) recursion;
search operators.

These function building operations are defined by straight-forward generalisations of the classical concepts on the natural numbers to concepts over a class K of many-sorted algebras whose domains include the natural numbers; such structures are called *standard algebras*. An important class COVIND(A) of functions on A is that of the *course-of-values (cov) inductively definable functions*, formed by combining sequential and parallel composition, course of values recursion, and least number search. If course-of-values recursion is replaced by primitive recursion in this definition then the class IND(A) of functions obtained is the class of inductively definable functions. In either case, the functions are defined by a *parallel deterministic* model of computation. The simultaneous recursions, which are responsible for this parallelism, are also required when computing on many-sorted structures. The basic definitions and theory are taken from Tucker and Zucker [88] in which it is argued in detail that whilst effective calculability is ill-defined as an informal idea when generalised to an abstract setting, the ideas of deterministic computation and operational semantics are meaningful and equivalent and, furthermore, the following is true:

Generalisation of the Church-Turing Thesis *Consider a deterministic programming language over an abstract data type D. The set of functions and relations on a structure A, representing an implementation of the abstract data type D, that can be programmed in the language,*

is contained in the set of cov inductively definable functions and relations on A. *The class of functions and relations over a class* K *of structures, representing a class of implementations of the abstract data type D, that can be programmed in the language, uniformly over all implementations of* K, *is contained in the class of cov inductively definable functions and relations over* K.

Much new work on translating recursions is necessary: see Simmons[88].

3.2 Computation on A*

For many purposes it is convenient to present the recursive functions on A using primitive recursion and the least number search operator on A*. This is possible because of the following fact

Lemma *Let f be a function on* A. *Then* $f \in$ COVIND(A) *if and only if* $f \in$ IND(A*).

4. SYNCHRONOUS CONCURRENT ALGORITHMS AND PARALLEL DETERMINISTIC COMPUTATION

A *synchronous concurrent algorithm* (sca) is an algorithm based on a network of modules and channels, computing and communicating data in parallel, and synchronised by a global clock. Synchronous algorithms process infinite streams of input data and return infinite streams of output data. Examples of scas include: *clocked hardware; systolic algorithms; neural nets; cellular automata;* and *coupled map lattice dynamical systems.*

4.1 A general functional model of synchronous concurrent computation.

To represent an algorithm, we first collect the sets A_i of data involved, and the functions f_i specifying the basic modules, to form a many sorted algebra A. To this algebra we adjoin a clock $T=\{0,1,...\}$ and the set $[T\rightarrow A_i]$ of *streams*, together with simple operations, to form a stream algebra $\underline{A}$ as in 2.5.1. This stream algebra defines the level of computational abstraction over which the sca is built.

An sca implements a specification that is a mapping of the form

$$F:[T\rightarrow A^n]\rightarrow[T\rightarrow A^m]$$

called a *stream transformer.*

The network and algorithm is then represented by means of the following method.

Suppose the algorithm consists of k modules and, for simplicity, that each module has several input channels, but only one output channel. Suppose that each model is connected to either other modules or the input streams.

Let us also suppose that each module produces an output at each clock cycle.

To each module m_i we associate a total function $V_i:T\times[T\rightarrow A^n]\times A^k\rightarrow A$ which defines the value $V_i(t,a,x)$ that is output from m_i at time t, if the algorithm is processing stream $a=(a_1, ...,a_k)$ from initial state $x=(x_1,...,x_k)$. The behaviour of the algorithm is represented by the parallel composition of functions $V_1, ..., V_k$.

More precisely, suppose that each module m_i has p(i) input channels, 1 output channel, and is specified by a function $f_i : A^{p(i)}\rightarrow A$. Suppose that the module m_i is connected to the modules $m_{\beta(i,1)},...,m_{\beta(i,p(i))}$ or to input streams $a_{\beta(i,1)},...,a_{\beta(i,p(i))}$.

We will assume that there is a delay along the channels that is specified by functions

$$\delta_{i,j} : T\times[T\rightarrow A^n]\times A^k\rightarrow T$$

which are subject to the condition that

$$\delta_{i,j}((t,a,x),a,x) < t$$

for all t,a,x. The maps V_i for $i=1,...,k$ are defined by the following:

For any i,

$$V_i(0,a,x) = x_i.$$

For i an input module,

$$V_i\,(t,a,x) = f\,(a_{\beta(i,1)}(\delta_{i,1}(t,a,x),a,x),\ldots,\ a_{\beta(i,p(i))}(\delta_{i,p(i)}(t,a,x),a,x))).$$

For i another module,

$$V_i\,(t,a,x) = f\,(V_{\beta(i,1)}(\delta_{i,1}(t,a,x),a,x),\ldots,\ V_{\beta(i,p(i))}(\delta_{i,p(i)}(t,a,x),a,x))).$$

This represents a *course-of-values recursion* on the stream algebra $\underline{A}$.

4.2 A constant delay model There are several simple conditions we may impose on the $\delta_{i,j}$ that directly reflect operational properties of the module, channels or network. For instance, we may assume that a fixed constant delay $d_{i,j}$ is assigned to each channel so that

$$\delta_{i,j}\,((t,a,x),a,x) \;= \min(t - d_{i,j},\, 0)$$

and the algorithm is given by these equations for the maps V_i for $i=1,\ldots,k$:

$$V_i\,(t,a,x) = x_i \quad \text{for } t < d_{i,j} + 1$$

$$V_i\,(t,a,x) = f\,(a_{\beta(i,1)}(t - d_{i,j},a,x),\ldots,\ a_{\beta(i,p(i))}(t - d_{i,j},a,x)))$$

$$V_i\,(t,a,x) = f\,(V_{\beta(i,1)}(t - d_{i,j},a,x),\ldots,\ V_{\beta(i,p(i))}(t - d_{i,j},a,x)))$$

4.3 A unit delay model If we take $d_{i,j} = 1$ then $\delta_{i,j}\,((t,a,x),a,x) \;= t - 1$ and we can rewrite the equations for the maps V_i for $i=1,\ldots k$:

$$V_i\,(0,a,x) = x_i$$

$$V_i\,(t+1,a,x) = f(a_{\beta(i,1)}\ (t,a,x),\ldots,\ a_{\beta(i,p(i))}\ (t,a,x)))$$

$$V_i\,(t+1,a,x) = f(V_{\beta(i,1)}\ (t,a,x),\ldots,\ V_{\beta(i,p(i))}\ (t,a,x)))$$

This is a *simultaneous primitive recursion over* $\underline{A}$.

Let us note that we have considered computation over a single algebra A and its stream algebra $\underline{A}$. *In practice* the above discussion invariably applies to a class of algebras. For example, often in the case of systolic algorithms, we design for the class of all initial (=standard) models of an axiomatisation of the integers or characters; or for some subclass of the class of all commutative rings.

4.4 Applications to hardware of the unit delay model Much of my research arisen from the aim to create a unified and comprehensive theory of hardware designs based on the concept of scaa and the methods and tools of algebra. To achieve this, B C Thompson and I have concentrated on the simple unit delay case which is already general enough to treat a huge number of interesting examples. The emphasis has been on case studies that evaluate practically applicable formal methods and software tools, and are useful in teaching. The programme can be divided into the following categories:

4.4.1 Models In addition to the functional model based on simultaneous recursive functions on $\underline{A}$, which is suited to work on specification and verification, we have considered other models in the unit delay case:

A von Neumann model based on concurrent assignments and function procedures on A; this is suited to work on programming, simulation and testing.

A directed graph model based on A; this is suited to work on architecture and layout.

Some models from each of these families have been formally defined by means of small languages, and, in particular, proved to be computationally equivalent. In formulating and classifying models

of synchronous computation we are following the pattern of work associated with computability theory and which ends with a Church Turing Thesis to establish the scope and limits of parallel deterministic models of computation. But, we are also motivated by consideration of a multirepresentational environment in which it is possible to intercompile between representations depending on one's work in the design of the algorithm.

See: Thompson and Tucker [85,88,89], Thompson [87], Meinke and Tucker [88] and Meinke [88].

4.4.2 Specification of scas and hardware. A substantial study of the specification of scas and their role in the process of designing hardware is underway. An important contibution is a very simple mathematical theory of clocks, and retimings between clocks, based on the following notions:

A *clock* is an algebra T=({0,1,2,...}| 0, t+1). A *retiming* of clock T to clock T' is a function r: T→ T' such that (i) r(0)=0; (ii) r is monotonic; and (iii) r is surjective.

Some theoretical results have been obtained on nonlinear retimings, hierarchical design, and synchronising clocks but the the main interest remains the application of the theoretical concepts in detailed case studies of the design of correlators and convolvers; counters; uarts; computers (including RISCII and VIPER).

The emphasis in this area is on the rigorous analysis of methodological models and practical formal methods. Very general methodological frameworks, based on formally defined notions of specifications as stream transformers over many sorted algebras, and their consistent refinement, have been developed.

See: Harman and Tucker [87,88a, 88b].

4.4.3 Derivation The systematic and formal derivation of scas have been studied in connection with rasterisation algorithms: see Eker and Tucker [87,88,89]. However, derivation is an area that requires further work. There is a large literature on developing systolic algorithms of various kinds, but much of it is *ad hoc*, informal, and application specific. Nevertheless research by H T Kung, P Quinton, and the more formal work of C Lengauer provide us with a platform on which to build an analysis of the algorithm design process that complements our analysis of the specification design process of mentioned in 4.4.2. Studies in the transformation of scas have been initiated in connection with a theoretical analysis of compilation of functional to graph descriptions. Using equational specifications for data types and term rewriting techniques, optimising transformations for scas have been defined as preprocessors to simple, but verified, compilers: see Meinke [88].

4.4.4 Verification of algorithms. The functional model is beautifully suited to verification and a substantial study of the verification of scas, based on that model, is well underway. A large number of case studies of hardware, and systolic algorithms (for linear algebra and string processing) have been verified: see Thompson and Tucker [85,89]; Hobley, Thompson and Tucker [88]; and Thompson [87].

In the light of this it is tempting to create independent software tools for verification, customised to our theories and techniques. However we see that the mathematical concepts and methods are of use in *many* existing approaches to machine assisted formal verification, including those of K Hanna, N Daeche and M Gordon (*HOL*, based on Church's type theory); R Constable (*Nuprl*, based on Martin Lof's type theory); and J Goguen (*OBJ*, based on term rewriting). Thus work with a number of existing theorem provers would be more useful for demonstrating the usefulness of our tools.

Of course, the logical foundations of the mathematical techniques are based on the use of many sorted first order logic found in Tucker and Zucker [88].

The recursive functions are closely related to equational logic and algebraic specification techniques based on initial algebra semantics. (and these in turn are easily related to Horn clauses and logic programming techniques). This is the subject of much research with J A Bergstra about the power of algebraic specification techniques to define computable algebras of various kinds: see Bergstra and Tucker [79,80,83,87], for example. With a general theory of computability the relevance of some of those ideas and techniques used in the proofs is revealed more clearly; and they are found to have practical application! As part of our work for a definitive paper on the unit delay model, B C Thompson and I have been using a generalisation of one of the simplest lemmas in Bergstra and Tucker [80] which was published as Bergstra and Tucker [87].

Let program algebra A_f be A augmented by all the subfunctions involved in the definition of f, obtained from its parse tree as a primitive recuirsive function in a certain straight forward way.

In the terminology of 2.2, A is the component algebra, A_f is the program algebra and (A,f) is the task algebra.

Let (Σ_f, E_f) be the signature and equations obtained by adding the corresponding names for these functions and equations obtained from the definition of the primitive recursive function f.

Theorem *Let* (Σ, E) *be an algebraic specification of the component algebra* A. *Let* f *be a primitive recursive function over* A. *Then the program algebra* $A_f \equiv I(\Sigma_f, E_f)$ *and hence* (Σ_f, E_f) *is an algebraic specification for the task algebra* (A, f).

Using a detailed proof of this fact, the functional definition of f over Σ can be mapped or compiled into an algebraic specification (Σ_f, E_f).

We can now work on the application of the proof of this result: we take scas, represented by primitive recursive functions over stream algebras, and map them into algebraic specifications, in preparation for machine processing.

Corollary *Let* (Σ, E) *be an algebraic specification of the stream algebra* $\underline{A}$. *Let* V*be primitive recursive over* $\underline{A}$, *representing an sca. Let program algebra* $\underline{A}_V$ *be* $\underline{A}$ *augmented by all subfunctions involved in the definition of V. Let* (Σ_V, E_V) *be the signature and equations corresponding with the primitive recursive function V.*

Then the program algebra $\underline{A}_V \equiv I(\Sigma_V, E_V)$ *and hence* (Σ_V, E_V) *is an algebraic specification for the sca algebra* (A, V).

(Some prototype programs have been constructed and applied to scas by B C Thompson.) This material will be included in a definitive paper on the unit delay model: Thompson and Tucker [89].

I may add that this machinery provides a huge number of algebraic specifictaions that are *not* related to the stack: clocked hardware; systolic arrays; cellular automata; neural nets, for instance!

4.4.5 Software tools. A detailed design of a programming language and programming environment based on the von Neumann model (ii) has been undertaken, and a prototype system has been constructed. This system includes: the language, which is called CARESS (for Concurrent Assignment REpresentation of Synchronous Systems); a C compiler; a preprocessor; and an interactive tracing/debugging tool. The prototype is robust and convenient enough to have been used in undergraduate teaching. A multilingual shell for animating, editing and debugging specifications of scas has been built. With this tool, it is possible to test specifications against their scas automatically. A test compiler from a functional notation for the recursive functions to Caress has been made.

4.5 Toward general theory of synchronous concurrent computation

The *Generalised Church-Turing Thesis* applies to delimit the class of computable functions over *any* algebra or class of algebras: the class is identified by COVIND(A) or IND(A*). Thus we have a tool to speed us toward the goal of establishing the scope and limits of synchronous computation in hardware by devising an appropriate generalisation of the Church-Turing Thesis. This synchronous computation is further identifed with the notion of *parallel deterministic computation over abstract data types with streams*. The class of stream transformations $F:[T \to A^n] \to [T \to A^m]$ is identified by their cartesian forms $F:[T \to A^n] \times T \to A^m$ in COVIND($\underline{A}$) or IND($\underline{A}$*).

However our interest in scas, and deep involvement with their applications, requires a comprehensive and indepently justifiable theoretical foundation. Thus full generalisations of the different models are needed to allow for more complex processing elements and timing characteristics; and these models must be compared with one-another and classified by constructing

compilers. A central problem is to understand *partiality* in terms of scas, which affects all aspects of the theory.

It is possible to model asynchronous nondeterministic computation in terms of synchronous deterministic computation in several ways. This was done for nondeterministic data flow by D Park, for example. The study of the nondeterminism and asynchrony as abstractions of determinism and synchrony is an important foundational task that has applications in practical modelling of hardware.

Extensive research on the functional model and its connections with equational specifications of modules, and with logic programming techniques, is necessary in order to support work on verification and software tools.

5. LOGIC PROGRAMMING AND SPECIFICATION

5.1 Specification and computation in logic programming A *logic programming language* is a language for specification *and* computation in which the means of computation is deduction in a logical system. More precisely, a program is a module that uses axiomatic theories expressed in formally defined logical languages, such as many-sorted first order logic, to define classes of implementation algebras (recall 2.1). This introduces the proof theory and as the basis for the semantics of computation; and, in particular, the model theory of logical systems as the basis for the semantics of specification. Unfortunately, the subject of proof dominates research on logic programming, mainly by work on practical deduction for implementations, and the subject of the model-theoretic semantics of specification has hardly been examined.

Clearly, to create modules we need theoretical accounts of constructing sets, defining relations, and evaluating functions. Thus research involves a range of programming paradigms and their integration: functional, relational, logic and, in its use of modules, object oriented. A relevant discussion is contained in Goguen and Meseguer [87].

5.2 Relational, functional and logic paradigms

The relational paradigm is for specification and the functional paradigm is for computation. The connection made in logic programming is of the following kind:

Let A be a set and R a subset of A^{n+m}. A *selection function* for R is a map $f:A^n \to A^m$ such that $\forall x[R(x,y) \Rightarrow R(x,f(x))]$.

The result of a logic program is the definition of a relation R and the computation of some family of selection functions for R. These constitute the task module of 2.2. A more appropriate general formulation is as follows:

Let T be a theory and let P be a logic program with goal relation R. Then we want to interpret P in a class K of models of T in order to specify relation R and compute selection functions f uniformly over some class K.

Thus we want to design P to be valid over a class K of algebras, and to compute one or more f such that:

$$K \models \forall x[R(x,y) \Rightarrow R(x,f(x))].$$

Note that this central problem of specifying relations and functions is a motivating problem of classical model theory: quantifier elimination. And that the problem of computing selection functions by logical deduction is a motivating problem of proof theory, and of the programs as proofs paradigm.

5.3 Scope and limits of specification

We have begun to study the corresponding characterisation of specification by means of relations. In Tucker and Zucker [89], the following basic question is asked and answered: *Does Horn clause computability on K, with its nondeterminism and potential for parallelism, specify all and only the cov inductively definable functions and relations on* K? The answer requires the basic step of formalising Horn clause computability over *any* structure or class of structures. It has been shown that Horn clause definability is fundamentally stronger than cov inductively definablility in general. It corresponds with an extension we have called *projective cov inductive definablility*.

In the computability theory of 3.1, we define a *semicomputable* set or relation R on A or K to be a set that is the domain of a partial computable function on A or K. It can be proved that *R is*

computable if and only if R and its complement -R is semicomputable.

A set or relation R is *projective semicomputable* if it is a projection of a semicomputable set: there is a computable function $f:A^n \times A^m \to A$ such that for all x, $R(x) \Leftrightarrow (\exists y \in A^m)[f(x,y)\downarrow]$.

Turning to course-of-values inductive definability, with Lemma in 3.2 in mind, we find:

Theorem *A relation R is definable by Horn clauses, uniformly over all A* in K* if, and only if, R is projective cov inductively semicomputable over K*.*

Not every set that is projective cov inductively semicomputable over A* is cov inductively semicomputable over A*. However, in the case of classes of *minimal structures*, Horn clause computability and cov semicomputability are equivalent.

These and other results begin to clarify the sense in which Horn clauses constitute a *specification language*, for it is possible to define Horn clause "programs" over certain models that cannot be executed deterministically. When axiomatic specifications of abstract data types are are employed in formal reasoning about programs it is not always possible to avoid such structures.

5.4 Other characterisations

Horn clause definability is equivalent to several other characterisations of nondeterministic models for specification including *while-array with initialisation; while-array with random assignments;* and older notions such as *search computability* in the sense of Y N Moschovakis, for example. J I Zucker and I are working on a generalisation of the Church-Turing Thesis for specification to complement that for computation in 3.1. This is a more complex task: for many more details see Tucker and Zucker [89].

5.5 Applications

This work is relevant to the development of the concept of a *logic programming module* that generalises the abstract data type module. There is a close connection between logic programming modules and algebraic specification modules: see Goguen and Meseguer [84], Tucker and Zucker [89] and Derrick, Fairtlough and Meinke[89]. This idea of a module is, of course, many sorted. Many sorted logic programming has been studied in depth by Walther[87] and Cohn[87], motivated by theorem prover efficiencies made possible by typing. See Derrick and Tucker [88] for a general discussions of these issues.

Acknowledgements

I would like to thank K Hobley, H Simmons, B C Thompson and S S Wainer for valuable conversations in the course of preparing this lecture.

6. REFERENCES

J W de Bakker, *Mathematical theory of program correctness*, Prentice Hall, 1980.

J.A. Bergstra and J V Tucker A characterisation of computable data types by means of a finite equational specification method, in J.W. de Bakker and J. van Leeuwen (eds.) *Automata, Languages and Programming, Seventh Colloquium, Noordwijkerhout, 1980,* Springer Lecture Notes in Computer Science 81, Springer-Verlag, Berlin, 1980, pp. 76-90.

J.A. Bergstra and J V Tucker Algebraic specifications of computable and semicomputable data structures, Research Report IW 121, Mathematisch Centrum, 1980.

J.A. Bergstra and J V Tucker Algebraic specifications of computable and semicomputable data types, *Theoretical Computer Science*, 50 (1987) 137-181.

J.A. Bergstra and J V Tucker, Initial and final algebra semantics for data type specifications: two characterisation theorems, *Society for Industrial and Applied Mathematics (SIAM) Journal on Computing*, 12 (1983) 366-387.

A G Cohn, A more expressive formulation of many sorted logic, *J. Automated Reasoning* 3 (1987) 113-200.

J Derrick, M Fairtlough and K Meinke, *Horn clause model theory and its applications in logic programming*, CTCS Report 1989.

J. Derrick and J V Tucker, Logic programming and abstract data types, In *Proceedings of 1988 UK IT Conference*, held under the auspices of the Information Engineering Directorate of the Department of Trade and Industry (DTI), Institute of Electrical Engineers (IEE), 217-219.

H Ehrig and B Mahr, *Fundamentals of algebraic specifications 1 - Equations and initial semantics*, Springer Verlag, 1985.

S M Eker and J V Tucker, Specification, derivation and verification of concurrent line drawing algorithms and architectures, pp 449-516 in *Theoretical foundations of computer graphics and CAD*, ed R A Earnshaw, Springer Verlag (1988).

S.M. Eker and J.V. Tucker, Specification and verification of synchronous concurrent algorithms : a case study of the Pixel Planes architecture. Centre for Theoretical Computer Science, University of Leeds, Report 12.88, 1988.
In P.M. Dew, R.A. Earnshaw and T.R. Heywood (eds), *Parallel processing for computer vision and display*, Addison Wesley, to appear.

J A Goguen, J W Thatcher, E G Wagner, and J B Wright, An initial algebra approach to the specification, correctness and implementation of abstract data types, pp 80-149 in *Current trends in programming methodology: IV Data structuring*, ed R T Yeh, Prentice Hall, 1978.

J A Goguen and J Meseguer, Equality, types, modules (and why not?) generics for logic programming, *J Logic Programming*, 2 (1984) 179-210.

J A Goguen and J Meseguer, Unifying functional, object-oriented and relational programming with logical semantics, Report SRI-CSL-87-7, SRI International, 1987.

N.A. Harman and J V Tucker, Clocks, retimings, and the formal specification of a UART, In G. Milne (ed) *The fusion of hardware design and verification* (Proceedings of IFIP Working Group 10.2 Working Conference), North-Holland, pp 375-396.

N.A. Harman and J.V. Tucker, The formal specification of a digital correlator I : User specification process.
Centre for Theoretical Computer Science, University of Leeds, Report 9.87, 1987.
In K. McEvoy and J.V. Tucker, *Theoretical aspects of VLSI design*, Cambridge University Press, to appear.

N.A. Harman and J V Tucker, Formal specifications and the design of verifiable computers, In *Proceedings of 1988 UK IT Conference*, held under the auspices of the Information Engineering Directorate of the Department of Trade and Industry (DTI), Institute of Electrical Engineers (IEE), pp 500-503.

K.M. Hobley, B.C. Thompson, and J V Tucker Specification and verification of synchronous concurrent algorithms: a case study of aconvolution algorithm, In G. Milne (ed.) *The fusion of hardware design and verification* (Proceedings of IFIP Working Group 10.2 Working Conference) North-Holland, pp 347-374.

J W Lloyd, *Foundations of Logic Programming*, Springer Verlag, 1984.

A I Mal'cev, *Algebraic systems*, Springer-Verlag, 1973.

A I Mal'cev, *The metamathematics of algebraic systems: Collected Papers 1936-1967,* North-Holland, 1971.

A.R. Martin and J.V. Tucker, The concurrent assignment representation of synchronous systems In J.W. de Bakker, A.J. Nijman and P.C. Treleaven (eds), *PARLE : Parallel Architectures and Languages Europe, Vol II Parallel languages*, Springer Lecture Notes in Computer Science 259, Springer-Verlag, 1987, 369-386.A revised and expanded edition is appears in *Parallel Computing* 9 (1988/89) 227-256.

K Meinke, A graph theoretic model of synchronous concurrent algorithms, PhD Thesis, School of Computer Studies, University of Leeds, Leeds, 1988.

K. Meinke and J.V. Tucker, Specification and representation of synchronous concurrent algorithms.
Centre for Theoretical Computer Science, University of Leeds, Report 22.88, 1988.
In F H Vogt (ed) *Concurrency* '88, Springer Lecture Notes in Computer Science, Springer-Verlag, 163-180.

K Meinke and J V Tucker, *Universal algebra* in S Abramsky, D Gabbay, T Maibaum (eds) *Handbook of logic in computer science*, OUP, to appear.

H Simmons, The realm of primitive recursion, *Archive Math Logic* 27 (1988) 117-188.

B.C. Thompson, A mathematical theory of synchronous concurrent algorithms.
PhD Thesis, School of Computer Studies, University of Leeds, 1987.

B.C. Thompson and J.V. Tucker, Theoretical considerations in algorithm design, In R.A. Earnshaw (ed), *Fundamental algorithms for computer graphics*, Springer-Verlag, 1985, 855-878.

B.C. Thompson and J V Tucker A parallel deterministic language and its application to synchronous concurrent algorithms, In *Proceedings of 1988 UK IT Conference*, held under the auspices of the Information Engineering Directorate of the Department of Trade and Industry (DTI), Institute of Electrical Engineers (IEE), pp 228-231.

B C Thompson and J V Tucker, Synchronous concurrent algorithms, in preparation, 1989.

J V Tucker and J I Zucker, *Program correctness over abstract data types, with error state semantics*, North Holland, Amsterdam, 1988.

J V Tucker and J I Zucker, Horn programs and semicomputable relations on abstract structures, Proceedings of the *Automata, Languages and Programming 1989, Sixteenth Colloquium, Stresa,* Springer Lecture Notes in Computer Science, Springer-Verlag, 1989.

C Walther, *A many sorted calculus based on resolution and paramodulation*, Pitman, 1987.

1964 - MC - J.W. de Bakker - CWI - 1989

door Loes Vasmel

KEYWORDS

zorgvuldig wetenschapper organisator voorzichtig

diplomatiek productief ordelijk introvert

irenisch sceptisch evenwichtig betrouwbaar

CHARACTERISTICS

To avoid misunderstandings

that I would also appreciate

However, I have to add to this that

let me recall from my letter of

a written confirmation of this

More specifically, I refer to

I therefore plan

On the other hand,

by return mail.

Allow me to point out to you

I would appreciate a quick reaction

RESOURCES

Angeline

Bas

Jaska

Catrien

Jacob

Lisa

Over ϕ's en ψ's

Hans van Vliet

Veel van semantiek weet ik niet, dus daar zal ik het niet over hebben. Wat rest zijn persoonlijke herinneringen aan onze "samenwerking" over een periode van ruim 20 jaar.

Ik ben op 1 september 1967 bij het toenmalige Mathematisch Centrum gaan werken, als operateur. Ik rolde papierband op, draaide aan de knoppen van de X1 en leerde programmeren in ALGOL 60.

Ook al werkte Jaco toen al geruime tijd bij het MC, onze eerste ontmoeting vond pas veel later plaats. Toen ik begon, droeg Jaco 's lands wapenrok. Tegen de tijd dat hij zulks lang genoeg had gedaan viel mij die eer te beurt.

Eind 1970 kwam ik uit militaire dienst. Niet lang daarna raakte ik als programmeur betrokken bij het ALGOL 68 project en werd Jaco mijn chef. Erg intensief waren onze contacten niet; programmeren deed (& doet) hij niet vaak. De fysieke afstand was ook vrij groot: hij in een opkamertje op de derde verdieping, wij in een niet al te frisse barak op de binnenplaats.

Ik maakte Jaco in een heel andere rol mee toen ik besloot informatica te gaan studeren aan de Vrije Universiteit (voor zover dat kon in 1973). Veel colleges heb ik niet gevolgd, als werkstudent. Toch ben ik vrij vaak bij Jaco's colleges, formele talen en semantiek van programmeertalen, geweest. Dit waren ook toen al notoire struikelblokken. Ik heb zelfs een van beide tentamens in één keer gehaald.

De, toen nog mondeling af te leggen, tentamens waren voor mij een martelgang. Ik heb slechts een eenvoudige hbs-b achtergrond en dus een navenant magere beheersing van het Griekse alfabet. Dat wreekt zich bij deze vakken. Tijdens zo'n tentamen werd ik geacht enkele bewijzen van stellingen te leveren. Ik schreef daarbij wat op het bord en gaf een mondelinge toelichting. Om de haverklap overkwam het mij hierbij dat ik ϕ schreef en ψ zei. Jaco corrigeerde mij dan heel geduldig. Mijn relatief lage cijfers voor deze vakken zullen wel niet aan dit manco kunnen worden toegeschreven, maar een uitgesproken liefde voor de theoretische informatica heeft het niet opgeleverd.

Vanuit een meer pragmatisch oogpunt had ik wel veel belangstelling voor formele talen. Ik heb daar zelfs ooit een stelling over bedacht, al wist ik niet goed hoe deze te bewijzen. Deze stelling ging over de relatie tussen LL(1) grammatica's en LR(1) grammatica's. Er bestaan voor beide typen grammatica's algoritmen om te controleren of een gegeven grammatica voldoet aan de hiervoor geldende eisen. Nu kan een algoritme om te controleren of een grammatica van type LR(1) is, ook gebruikt worden om te controleren of een grammatica van type LL(1) is, en wel als volgt:

> Als een grammatica G van type LL(1) is, dan is de grammatica G' van type LR(1), waarbij G' uit G wordt verkregen door elke produktieregel van de vorm $A \rightarrow \alpha$ in G te vervangen door een produktieregel van de vorm $A \rightarrow N_i\alpha$ in G'. Hierbij moet tevens aan G' nog een voldoend aantal produktieregels van de vorm $N_i \rightarrow \varepsilon$ worden toegevoegd.

Intuïtief is dit als volgt in te zien. Bij een grammatica van type LR(1) is één symbool voorbij het eind van een produktieregel te beslissen of die regel van toepassing is. Bij een grammatica van type LL(1) is dit na het eerste symbool van die produktieregel al vast te stellen. Als G' van type LR(1) is, is één symbool voorbij het voorkomen van N_i te beslissen of deze produktieregel van toepassing is. Aangezien alle N_i verschillend, en leeg, zijn, is zulks na het eerste symbool van α te beslissen. Maar dan is G van type LL(1).

Ik was van plan deze stelling bij mijn proefschrift op te nemen, maar moest natuurlijk wel weten of niet iemand anders dit al lang ook bedacht had. Jaco kon mij hierbij niet helpen. Anton Nijholt wel. Inderdaad was deze stelling al eerder bedacht door iemand anders.

Ooit is het nog bijna tot een echte samenwerking tussen ons beiden gekomen. Begin september 1982 kregen wij de vraag voorgelegd om in het kader van de pilot-fase van ESPRIT een project in te dienen. Zoals gebruikelijk was er veel haast bij. Jaco belde mij thuis op met de vraag of ik in het weekend mee wilde werken aan het opstellen van een voorstel. Ik heb dat niet gedaan; ik was jarig en vond dat belangrijker dan ESPRIT. Het voorstel is er toch gekomen ("formele specificatie van programmatuur voor gedistribueerde berekeningen"), maar is uiteindelijk niet gehonoreerd.

Inmiddels ben ik niet meer in dienst van het CWI. Contact hebben we nog steeds, zij het nu als collega's. Deze contacten zijn ook nu niet erg intensief, maar nog immer heel plezierig.

Dear Jaco,

Congratulations on the 25th anniversary of your stay at the CWI, during which time the subject of program semantics has changed beyond recognition, due in significant measure to your own work.

I recall with great pleasure my own stay at the CWI in 1978-79, and my continued association with your group since then.

Wishing you many more years of continued success,

Jeff Zucker
Buffalo, New York, USA

March 1989