CWI Quarterly Volume 11 (2&3) 1998, pp. 299 – 321

Unfold/Fold Transformations of CCP Programs [1]

Sandro Etalle
Universiteit Maastricht, P.O. Box 616, 6200 MD Maastricht, The Netherlands.
etalle@cs.unimaas.nl.

Maurizio Gabbrielli
Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy.
gabbri@di.unipi.it.

Maria Chiara Meo
Università di L'Aquila, Via Vetoio, Loc. Coppito, 67010 L'Aquila, Italy.
meo@univaq.it.

We introduce a transformation system for concurrent constraint programming (CCP). We define suitable applicability conditions for the transformations which guarantee that the input/output CCP semantics is preserved also when distinguishing deadlocked computations from successful ones.

The system allows to optimize CCP programs while preserving their intended meaning. Furthermore, since it preserves the deadlock behaviour of programs, it can be used for proving deadlock freeness of a class of queries in a given program.

Keywords: Transformation, Concurrent Constraint Programming.

1. Introduction

Optimization techniques, in the case of logic-based languages, fall into two main categories: on one hand, there exist methods for compile-time and low-level optimizations such as the ones presented for constraint logic programs in [11], which are usually based on program analysis methodologies (e.g. abstract interpretation). On the other hand, we find source to source transformation

[1] The three authors have simultaneously been researchers/visiting researchers at CWI, where their research on transformations of constraint logic languages began. A preliminary version of this paper appeared in [8].

techniques such as *partial evaluation* (see [17]) (which in the field of logic programming is mostly referred to as *partial deduction* and is due to Komorowski [13]), and more general techniques based on the *unfold* and *fold* or on the *replacement* operation.

Unfold/fold transformation techniques were first introduced for functional programs in [2], and then adapted to logic programming (LP) both for program synthesis [3, 10], and for program specialization and optimization [13]. Tamaki and Sato in [24] proposed a general framework for the unfold/fold transformation of logic programs, which has remained in the years the main historical reference of the field, and has recently been extended to constraint logic programming (CLP) in [1, 5, 15] (for an overview of the subject, see the survey by Pettorossi and Proietti [18]). As shown by a number of applications, these techniques provide powerful methods for the development and optimization of large programs, and can be regarded as the *basic* transformations techniques, which might be further adapted to be used for partial evaluation.

Despite a large literature in the field of sequential languages, unfold/fold transformation sequences have hardly been applied to concurrent logic languages. Notable exceptions are the papers of Ueda and Fukurawa [25], Sahlin [19], and of de Francesco and Santone [9] (the relations with this paper are discussed in Section 5). This situation is partially due to the fact that the non-determinism and the synchronization mechanisms present in concurrent languages substantially complicate their semantics, thus complicating also the definition of *correct* transformation systems. Nevertheless, as argued below, transformation techniques can be be more useful for concurrent languages than they already are for sequential ones.

In this paper we introduce a transformation system for concurrent constraint programming (CCP) [20, 21, 22]. This paradigm derives from replacing the *store-as-valuation* concept of von Neumann computing by the *store-as-constraint* model: Its computational model is based on a global *store*, which consists of the conjunction of all the constraints established until that moment and expresses some partial information on the values of the variables involved in the computation. Concurrent processes synchronize and communicate asynchronously via the store by using elementary actions (ask and tell) which can be expressed in a logical form (essentially implication and conjunction [4]). On the one hand, CCP enjoys a clean logical semantics, avoiding many of the complications arising in the concurrent imperative setting; as argued in the position paper [6] this aspect is of great help in the development of effective transformation (and partial evaluation) tools. On the other hand, CCP benefits from a number of existing implementations, an example being Oz [23]; thus, in contrast to other models for concurrency such as the π-calculus, in this framework transformation techniques can be readily applied to practical problems.

The transformation system we are going to introduce is originally inspired by the system of Tamaki and Sato [24], on which it improves in three main ways: firstly, by taking full advantage of the flexibility and expressivity of CCP, it

introduces a number of new important transformation operations, allowing optimizations that would not be possible in the LP or CLP context; secondly, our system we managed to eliminate the limitation that in a folding operation the *folding clause* has to be nonrecursive, a limitation which is present in virtually all other unfold/fold transformation systems, this improvement possibly leads to the use of new more sophisticated transformation strategies; finally, the applicability conditions we propose for the folding operation are now independent from the *transformation history*, making the operation much easier to understand and, possibly, to be implemented.

We will illustrate with a practical example how our transformation system for CCP can be even more useful than its predecessors for sequential logic languages. Indeed, in addition to the usual benefits, in this context the transformations can also lead to the elimination of communication channels and of synchronization points, to the transformation of non-deterministic computations into deterministic ones, and to the crucial saving of computational *space*. It is also worth mentioning that the declarative nature of CCP allows us to define reasonably simple applicability conditions which ensure the correctness of our system.

Our results show that the original and the transformed program have the same input/output behaviour both for successful and for deadlocked derivations. As a corollary, we obtain that the original program is deadlock free iff the transformed one is, and this allows to employ the transformation as an effective tool for proving deadlock-freeness: if, after the transformation, we can prove or see that the process we are considering never deadlocks (in some cases the transformation simplifies the program's behaviour so that this can be immediately checked), then we are also sure that the original process does not deadlock either.

This paper is organized as follows: in the next section we present the notation and the necessary preliminary definitions, most of them regarding the CCP paradigm. In Section 3 we define the transformation system, which consists of various different operations and for this reason the section is divided in a number of subsections. Section 4 states the main result, concerning the correctness of the transformation system, while Section 5 concludes by comparing this paper to related work in the literature. Proof sketches are given in the Appendix.

2. Preliminaries

The basic idea underlying CCP is that computation progresses via monotonic accumulation of information in a global store. Information is produced by the concurrent and asynchronous activity of several agents which can *add* a constraint c to the store by performing the basic action $\mathsf{tell}(\mathsf{c})$. Dually, agents can also *check* whether a constraint c is entailed by the store by using an $\mathsf{ask}(\mathsf{c})$ action. This allows the synchronization of different agents.

Concurrent constraint languages are defined parametrically wrt to the no-

tion of *constraint system*, which is usually formalized in an abstract way and is provided along with the guidelines of Scott's treatment of information systems (see [21]). Here, we consider a more concrete notion of constraint which is based on first-order logic and which coincides with the one used for constraint logic programming. This will allow us to define the transformation operations in a more comprehensible way, while retaining a sufficient expressive power. Thus a *constraint* c is a first-order formula built by using predefined predicates (called primitive constraints, which always include equality) over a computational domain $\mathcal{D}$. Formally, $\mathcal{D}$ is a *structure* which determines the interpretation of the constraints.

In the sequel, terms will be denoted by $\mathsf{t}, \mathsf{s}, \ldots$, variables with $\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots$, further, as a notational convention, $\tilde{\mathsf{t}}$ and $\tilde{\mathsf{X}}$ denote a tuple of terms and a tuple of distinct variables, respectively. $\exists_{-\tilde{\mathsf{X}}}\ \mathsf{c}$ stands for the existential closure of c *except* for the variables in $\tilde{\mathsf{X}}$ which remain unquantified. The formula $\mathcal{D} \models \exists_{-\tilde{\mathsf{X}}}\mathsf{c}$ states that $\exists_{-\tilde{\mathsf{X}}}\mathsf{c}$ is valid in the interpretation provided by $\mathcal{D}$, i.e. that it is true for every binding of the free variables of $\exists_{-\tilde{\mathsf{X}}}\mathsf{c}$. The empty conjunction of primitive constraints will be identified with true. We also denote by $Var(\mathsf{e})$ the set of variables occurring in the expression e.

The notation and the semantics of programs and agents is virtually the same one of [21]. In particular, the $\|$ operator allows one to express parallel composition of two agents and it is usually described in terms of interleaving, while non-determinism arises by introducing a (global) choice operator $\sum_{i=1}^{n} \mathsf{ask}(\mathsf{c}_i) \rightarrow \mathsf{A}_i$: the agent $\sum_{i=1}^{n} \mathsf{ask}(\mathsf{c}_i) \rightarrow \mathsf{A}_i$ nondeterministically selects one $\mathsf{ask}(\mathsf{c}_i)$ which is enabled in the current store, and then behaves like A_i. Thus, the syntax of CCP *declarations* and *agents* is given by the following grammar:

Declarations	$\mathsf{D} ::=$	$\epsilon \mid \mathsf{p}(\tilde{\mathsf{t}}) \leftarrow \mathsf{A} \mid \mathsf{D}, \mathsf{D}$
Agents	$\mathsf{A} ::=$	$\mathsf{stop} \mid \mathsf{tell}(\mathsf{c}) \mid \sum_{i=1}^{n} \mathsf{ask}(\mathsf{c}_i) \rightarrow \mathsf{A}_i \mid \mathsf{A} \,\|\, \mathsf{A} \mid \mathsf{p}(\tilde{\mathsf{t}})$
Processes	$\mathsf{Proc} ::=$	$\mathsf{D}.\mathsf{A}$

where c and c_i's are constraints. Note that, differently from [21], here we allow terms as arguments to predicate symbols. Due to the presence of an explicit choice operator, as usual we assume that each predicate symbol is defined by exactly one declaration. A *program* is a set of declarations.

An important aspect for which we slightly depart from the usual formalization of CCP regards the notion of *locality*. In [21] locality is obtained by using the operator $\exists$, and the behaviour of the agent $\exists_{\mathsf{X}}\,\mathsf{A}$ is defined like the one of A, with the variable X considered as *local* to it. Here we do not use such an explicit operator: analogously to the standard CLP setting, locality is introduced implicitly by assuming that if a process is defined by $\mathsf{p}(\tilde{\mathsf{X}}) \leftarrow \mathsf{A}$ and a variable Y occurs in A but not in $\tilde{\mathsf{X}}$, then Y has to be considered local to A.

The operational model of CCP is described by a transition system $\mathsf{T} = (\mathsf{Conf}, \rightarrow)$ where configurations (in) Conf are pairs consisting of a process and a constraint (representing the common *store*), while the transition relation

R1	$\langle D.\mathsf{tell}(c), d\rangle \rightarrow \langle D.\mathsf{stop}, c \wedge d\rangle$	
R2	$\langle D.\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i, d\rangle \rightarrow \langle D.A_j, d\rangle$	if $j \in [1,n]$ *and* $\mathcal{D} \models d \rightarrow c_j$
R3	$\dfrac{\langle D.A, c\rangle \rightarrow \langle D.A', c'\rangle}{\begin{array}{l}\langle D.(A \parallel B), c\rangle \rightarrow \langle D.(A' \parallel B), c'\rangle \\ \langle D.(B \parallel A), c\rangle \rightarrow \langle D.(B \parallel A'), c'\rangle\end{array}}$	
R4	$\langle D.p(\tilde{t}), c\rangle \rightarrow \langle D.A \parallel \mathsf{tell}(\tilde{t} = \tilde{s}), c\rangle$	if $p(\tilde{s}) \leftarrow A \in \mathsf{defn}_D(p)$

TABLE 1. The (standard) transition system.

$\rightarrow \subseteq \mathsf{Conf} \times \mathsf{Conf}$ is described by the (least relation satisfying the) rules **R1**-**R4** of Table 1 which should be self-explaining. Here and in the following we assume given a program D and we denote by $\mathsf{defn}_D(p)$ the set of variants[2] of the definition in D for the predicate symbol p. Due to the presence of terms as arguments to predicates symbols, differently from [21], in rule **R4** parameter passing is performed by an explicit tell action. We assume also the presence of a renaming mechanism that takes care of using fresh variables each time a declaration is considered[3].

We denote by $\rightarrow^*$ the reflexive-transitive closure of the relation $\rightarrow$ defined by the transition system, and we denote by Stop any agent which contains only stop and $\parallel$ constructs. A finite derivation (or computation) is called *successful* if it is of the form $\langle D.A, c\rangle \rightarrow^* \langle D.\mathsf{Stop}, d\rangle \not\rightarrow$ while it is called *deadlocked* if it is of the form $\langle D.A, c\rangle \rightarrow^* \langle D.B, d\rangle \not\rightarrow$ with B different from Stop (i.e., B contains at least one suspended agent). Note that we consider here the so called "eventual tell" CCP, i.e. when adding constraints to the store (via tell operations) there is no consistency check.

Using the transition system in Table 1 we define the notion of observables as follows. Here and in the sequel we say that a constraint c is *satisfiable* iff $\mathcal{D} \models \exists\ c$.

DEFINITION 2.1 (OBSERVABLES) *Let* $D.A$ *be a CCP process. We define*

[2] A variant of a declaration d is obtained by replacing the tuple $\tilde{X}$ of all the variables appearing in d for another tuple $\tilde{Y}$.

[3] For the sake of simplicity we do not describe this renaming mechanism in the transition system. The interested reader can find in [21, 22] various formal approaches to this problem.

$$\mathcal{O}(\mathsf{D.A}) = \{\langle \mathsf{c}, \exists_{\_\mathsf{Var(A,c)}}\mathsf{d}, \mathsf{ss}\rangle \mid \mathsf{c} \text{ and } \mathsf{d} \text{ are satisfiable, and there exists a derivation } \langle \mathsf{D.A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D.Stop}, \mathsf{d}\rangle\}$$
$$\cup$$
$$\{\langle \mathsf{c}, \exists_{\_\mathsf{Var(A,c)}}\mathsf{d}, \mathsf{dd}\rangle \mid \mathsf{c} \text{ and } \mathsf{d} \text{ are satisfiable, and there exists a derivation } \langle \mathsf{D.A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D.B}, \mathsf{d}\rangle \not\rightarrow, B \neq Stop\}$$

□

Thus what we observe are the results of finite computations (if consistent), abstracting from the values for the local variables in the results, and distinguishing the successful computations from the deadlocked ones (by using the termination modes ss and dd, respectively). This provides the intended semantics to be preserved by the transformation system: we will call a transformation *correct* if it maps a program into another one having the same observables; given the above definition, this will allow us to compare with each other the "deadlocks" and the "successes" of the original and the transformed programs.

3. The Transformation

In order to illustrate the application of our method we'll adopt a working example. We consider an auction problem in which two bidders participate: bidder_a and bidder_b; each bidder takes as input the list of the bids of the other one and produces as output the list of his own bids. When one of the two bidders wants to quit the auction, it produces in its own output stream the token quit. This protocol is implemented by the following program AUCTION.

```
auction(LeftBids,RightBids) ←
        bidder_a([0|RightBids],LeftBids) ||bidder_b(LeftBids,RightBids)


bidder_a(HisList, MyList) ←
   ( ask(∃HisBid,HisList' HisList = [HisBid|HisList'] ∧ HisBid = quit) → stop
   + ask(∃HisBid,HisList' HisList = [HisBid|HisList'] ∧ HisBid ≠ quit) →
        tell(HisList = [HisBid|HisList']) ||
        make_new_bid_a(HisBid,MyBid) ||
        ( ask(MyBid = quit) → tell(MyList = [MyBid|MyList']) ||
                              broadcast("a quits")
        + ask(MyBid ≠ quit) → tell(MyList = [MyBid|MyList']) ||
             tell(MyBid ≠ quit) ||
             bidder_a(HisList',MyList')))
```

plus an analogous definition for bidder_b [4].

[4] In the above program the agent tell(HisList = [HisBid|HisList']) is needed to bind the local variables (HisBid, HisList') to the global one (HisList): In fact, as resulting from the operational semantics, such a binding is not performed by the ask agent. On the contrary the agent tell(MyBid ≠ quit) is redundant: We have introduced it in order to simplify the following transformations. Actually this introduction of redundant tells is a transformation operation which is omitted here for space reasons.

Here, the agent make_new_bid_a(HisBid,MyBid) is in charge of producing a new offer in presence of the competitor's offer HisBid; the agent will produce MyBid = quit if it evaluates that HisBid is too high to be topped, and decides to leave the auction. Notice that in order to avoid deadlock, auction initializes the auction by inserting a fictitious zero bid in the input of bidder a.

3.1. Introduction of a new definition

The introduction of a new definition is virtually always the first step of a transformation sequence. Since the new definition is going to be the main target of the transformation operation, this step will actually determine the very direction of the subsequent transformation, and thus the degree of its effectiveness.

Determining which definitions should be introduced is a very difficult task which falls into the area of *strategies*. To give a simple example, if we wanted to apply *partial evaluation* to our program with respect to a given agent A (i.e. if we wanted to specialize our program so that it would execute the partially instantiated agent A in a more efficient way), then a good starting point would most likely be the introduction of the definition $p(\tilde{X}) \leftarrow A$, where $\tilde{X}$ is an appropriate tuple of variables and p is a new predicate symbol. Now, a different strategy would probably determine the introduction of a different new definition. For a survey of the other possibilities we refer to [18].

In this paper we are not concerned with the strategies, but only with the basic transformation operations and their correctness: we aim at defining a transformation system which is general enough so to be applied in combination with different strategies. In order to simplify the terminology and the technicalities, we assume that these new declarations are added once for all to the original program before starting the transformation itself. Note that this is clearly not restrictive. As a notational convention we call D_0 the program obtained after the introduction of new definitions. In the case of program AUCTION, we assume that the following new declarations are added to the original program.

auction_left(LastBid) $\leftarrow$ tell(LastBid $\neq$ quit) || bidder_a([LastBid|Bs],As) || bidder_b(As,Bs).
auction_right(LastBid) $\leftarrow$ tell(LastBid $\neq$ quit) || bidder_a(Bs,As) || bidder_b([LastBid|As],Bs).

The agent auction_left(LastBid) engages an auction starting from the bid LastBid (which cannot be quit) and expecting the bidder "a" to be the next one in the licit. The agent auction_right(LastBid) is symmetric.

3.2. Unfolding

The first transformation we consider is the *unfolding*. This operation consists essentially in the replacement of a procedure call by its definition. The syntax of CCP agents allows us to define it in a very simple way by using the notion of context. A *context*, denoted by C[ ], is simply an agent with a "hole". C[A] denotes the agent obtained by replacing the hole in C[ ] for the agent A, in the obvious way.

DEFINITION 3.1 (UNFOLDING) *Consider a set of declarations* D *containing*

$\mathsf{d}: \quad \mathsf{H} \leftarrow \mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})]$

$\mathsf{u}: \quad \mathsf{p}(\tilde{\mathsf{s}}) \leftarrow \mathsf{B}$

Then unfolding $\mathsf{p}(\tilde{\mathsf{t}})$ *in* d *consists in replacing* d *by*

$\mathsf{d}': \quad \mathsf{H} \leftarrow \mathsf{C}[\mathsf{B} \parallel \mathsf{tell}(\tilde{\mathsf{s}} = \tilde{\mathsf{t}})]$

in D. *Here* d *is the* unfolded *definition and* u *is the unfolding one;* d *and* u *are assumed to be renamed so that they do not share variables.* □

After an unfolding we often need to evaluate some of the newly introduced tell's in order to "clean up" the resulting declarations. To this aim we introduce the following operation. Here we assume that the reader is acquainted with the notion of *substitution* and of (relevant) *most general unifier* (see [14]). We denote by $e\sigma$ result of the application of a substitution σ to an expression e.

DEFINITION 3.2 (TELL EVALUATION) *A declaration*

$\mathsf{d}: \quad \mathsf{H} \leftarrow \mathsf{C}[\mathsf{tell}(\tilde{\mathsf{s}} = \tilde{\mathsf{t}}) \parallel \mathsf{B}]$

is transformed by tell evaluation to

$\mathsf{d}': \quad \mathsf{H} \leftarrow \mathsf{C}[\mathsf{B}\sigma]$

where σ *is a relevant most general unifier of* s *and* t, *and the variables in the domain*[5] *of* σ *do not occur neither in* $\mathsf{C}[\,]$ *nor in* H. □

These applicability conditions can in practice be weakened by appropriately renaming some local variables. In fact, if all the occurrences of a local variable in $\mathsf{C}[\,]$ are in choice branches different from the one the "hole" lies in, then we can safely rename apart each one of these occurrences.

In our AUCTION example, we start working on the definition of auction_right, and we unfold the agent bidder_b([LastBid|As], Bs) and then we perform the subsequent tell evaluations. The result of these operations is the following program.

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
   bidder_a(Bs, As) ||
   ( ask(∃HisBid,HisList' [LastBid|As] = [HisBid|HisList'] ∧ HisBid = quit) → stop
   + ask(∃HisBid,HisList' [LastBid|As] = [HisBid|HisList'] ∧ HisBid ≠ quit) →
        tell([LastBid|As] = [HisBid|HisList']) ||
        make_new_bid_b(HisBid,MyBid) ||
        ( ask(MyBid = quit) → tell(Bs = [MyBid|Bs']) || broadcast("b quits")
        + ask(MyBid ≠ quit) → tell(Bs = [MyBid|Bs']) ||
             tell(MyBid ≠ quit) ||
             bidder_b(HisList',Bs')))
```

[5] We recall that, given a substitution σ, the domain of σ is the finite set of variables $\{\mathsf{X} \mid \mathsf{X}\sigma \neq \mathsf{X}\}$.

Another new operation, similar to the one of unfolding, is the one of *backward instantiation.*

DEFINITION 3.3 (BACKWARD INSTANTIATION) *Let* D *be a set of definitions and*

$$\begin{array}{ll} \mathsf{d}: & \mathsf{H} \leftarrow \mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})] \\ \mathsf{b}: & \mathsf{p}(\tilde{\mathsf{s}}) \leftarrow \mathsf{tell}(\mathsf{c}) \parallel \mathsf{B} \end{array}$$

be two definitions of D. *Suppose also that* c' *is a constraint such that* $\mathcal{D} \models \mathsf{c} \rightarrow \mathsf{c}'$. *Then the* backward instantiation *of* $\mathsf{p}(\tilde{\mathsf{t}})$ *in* d *via* c' *consists in replacing* d *by*

$$\mathsf{d}': \quad \mathsf{H} \leftarrow \mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}}) \parallel \mathsf{tell}(\mathsf{c}') \parallel \mathsf{tell}(\tilde{\mathsf{t}} = \tilde{\mathsf{s}})]$$

(it is assumed here that d *and* b *are renamed so that they have no variables in common).*

The operation can also be applied when b *is not of the form* $\mathsf{p}(\tilde{\mathsf{s}}) \leftarrow \mathsf{tell}(\mathsf{c}) \parallel \mathsf{B}$ *by considering* c *to be* true. □

Intuitively, this operation can be regarded as a "half-unfolding" for the following reason: performing an unfolding is equivalent to applying a derivation step to the atomic agent under consideration, here we don't quite do it, yet we carry out (part of) the two first phases that the derivation step requires.

3.3. Guard Simplification

A new important operation is the one which allows us to modify the ask guards occurring in a program. Consider an agent of the form $\mathsf{C}[\mathsf{ask}(\mathsf{c}) \rightarrow \mathsf{A} + \mathsf{ask}(\mathsf{d}) \rightarrow \mathsf{B}]$ and a given set of declarations. Let us call *weakest produced constraint* of $\mathsf{C}[\,]$ the conjunction of all the constraints appearing in ask and tell actions which certainly have to be evaluated before $[\,]$ is reached (in the context $\mathsf{C}[\,]$). Now, if a is the weakest produced constraint of $\mathsf{C}[\,]$ and $\mathcal{D} \models \mathsf{a} \rightarrow \mathsf{c}$ then clearly we can simplify the previous agent to $\mathsf{C}[\mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{A} + \mathsf{ask}(\mathsf{d}) \rightarrow \mathsf{B}]$[6]. In general, if a is the context constraint of $\mathsf{C}[\,]$, and for some constraint c' we have that $\mathcal{D} \models \exists_{-\tilde{\mathsf{z}}}\ (\mathsf{a} \wedge \mathsf{c}) \leftrightarrow (\mathsf{a} \wedge \mathsf{c}')$ (where $\tilde{\mathsf{z}} = \mathit{Var}(\mathsf{C}, \mathsf{A})$), then we can replace c with c'. In particular, if we have that $\mathsf{a} \wedge \mathsf{c}$ is unsatisfiable, then c can immediately be replaced with false (the unsatisfiable constraint). In order to formalize this intuitive idea, we start with the following definition.

DEFINITION 3.4. *Let* D *be a (fixed) set of declarations, and* s *be a set of predicates. Given an agent* A, *its* weakest produced constraint *(with respect to* s*), is denoted by* $\mathsf{wpc}_{\mathsf{s}}(\mathsf{A})$ *and is defined by structural induction as follows:*

[6] Note also that in general the further simplification to $\mathsf{C}[\mathsf{A} + \mathsf{ask}(\mathsf{d}) \rightarrow \mathsf{B}]$ is not correct, while we can transform $\mathsf{C}[\mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{A}]$ into $\mathsf{C}[\mathsf{A}]$.

$$
\begin{array}{rcl}
\mathsf{wpc_s}(\mathsf{stop}) & = & \mathsf{true} \\
\mathsf{wpc_s}(\mathsf{tell}(\mathsf{c})) & = & \mathsf{c} \\
\mathsf{wpc_s}(\mathsf{A} \parallel \mathsf{B}) & = & \mathsf{wpc_s}(\mathsf{A}) \wedge \mathsf{wpc_s}(\mathsf{B}) \\
\mathsf{wpc_s}(\sum_{\mathsf{i}} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i}) & = & \mathsf{true} \\
\mathsf{wpc_s}(\mathsf{p}(\tilde{\mathsf{t}})) & = & \begin{cases} \mathsf{wpc}_{(\mathsf{s} \cup \{\mathsf{p}\})}(\mathsf{A}) & \text{if } \mathsf{p} \notin \mathsf{s} \text{ and} \\ & \quad \mathsf{p}(\tilde{\mathsf{t}}) \leftarrow \mathsf{A} \in \mathsf{defn_D}(\mathsf{p}(\tilde{\mathsf{t}})) \\ \mathsf{true} & \text{if } \mathsf{p} \in \mathsf{s} \end{cases}
\end{array}
$$

s *contains then the set of predicates which should not be taken into consideration. Given a context* C[ ] *and a set of predicate symbols* s *the* weakest produced constraint*, of* C[ ] *(with respect to* s*)* $\mathsf{wpc_s}(\mathsf{C}[\,])$*, is inductively defined as follows:*

$\mathsf{wpc_s}([\,]) = \mathsf{true}$
$\mathsf{wpc_s}(\mathsf{C}'[\,] \parallel \mathsf{B}) = \mathsf{wpc_s}(\mathsf{B}) \wedge \mathsf{wpc_s}(\mathsf{C}'[\,])$
$\mathsf{wpc_s}(\sum_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i}) = \mathsf{c_j} \wedge \mathsf{wpc_s}(\mathsf{C}'[\,])$ *where* $\mathsf{j} \in [1, \mathsf{n}]$ *and* $\mathsf{A_j} = \mathsf{C}'[\,]$

Notice that the weakest produced constraint depends on the set of declarations D under consideration. We are now ready to define the operation of guard simplification.

DEFINITION 3.5 (GUARD SIMPLIFICATION) *Let* D *be a set of declarations, and*

$$\mathsf{d}: \ \mathsf{H} \leftarrow \mathsf{C}[\textstyle\sum_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i}]$$

be a declaration of D*. Assume that for some constraints* $\mathsf{c}'_1, \ldots, \mathsf{c}'_{\mathsf{n}}$ *we have that for* $j \in [1, n]$*,*

$$\mathcal{D} \models \exists_{-\tilde{\mathsf{z}}_{\mathsf{j}}} (\mathsf{wpc}_{\emptyset}(\mathsf{C}[\,]) \wedge \mathsf{c_j}) \leftrightarrow (\mathsf{wpc}_{\emptyset}(\mathsf{C}[\,]) \wedge \mathsf{c}'_{\mathsf{j}}) \quad (\textit{where } \tilde{\mathsf{z}}_{\mathsf{j}} = \mathit{Var}(\mathsf{C}, \mathsf{H}, \mathsf{A_j})),$$

then we can replace d *with*

$$\mathsf{d}': \ \mathsf{H} \leftarrow \mathsf{C}[\textstyle\sum_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{ask}(\mathsf{c}'_{\mathsf{i}}) \rightarrow \mathsf{A_i}] \qquad \square$$

In our AUCTION example, we can consider the weakest produced constraint of tell(LastBid $\neq$ quit), and modify the subsequent ask constructs as follows

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
   bidder_a(Bs, As) ||
      ask(∃HisBid,HisList' [LastBid|As] = [HisBid|HisList']
                           ∧ LastBid ≠ quit ∧ HisBid = quit) →
         stop
   + ask(∃HisBid,HisList' [LastBid|As] = [HisBid|HisList']) →
         tell([LastBid|As] = [HisBid|HisList']) ||
         ...
```

Via the same operation, we can immediately simplify this to.

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || bidder_a(Bs, As) ||
      ask(false) → stop
   + ask(true) → tell([LastBid|As] = [HisBid|HisList']) ||
         ...
```

Branch Elimination and Conservative Guard Evaluation Notice that in the above program, we have a guard ask(false) which of course will never be satisfied. The first important application of the guard simplification operation regards then the elimination of unreachable branches.

DEFINITION 3.6 (BRANCH ELIMINATION) *Let*

$$\mathsf{d}: \ \mathsf{H} \leftarrow \mathsf{C}[\sum_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i}]$$

be a declaration. Assume that $n > 1$ and that for some $j \in [1, n]$, we have that $\mathsf{c_j} \equiv \mathsf{false}$*, then we can replace* d *with*

$$\mathsf{d}': \ \mathsf{H} \leftarrow \mathsf{C}[(\sum_{\mathsf{i}=1}^{\mathsf{j}-1} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i}) \ + \ (\sum_{\mathsf{i}=\mathsf{j}+1}^{\mathsf{n}} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i})] \qquad \square$$

The condition that $n > 1$ ensures that we are not eliminating all the branches (if we wanted to do so, and of course if we were allowed to, that is, if all the guards are unsatisfiable, then we could do so by replacing the whole choice with a new special agent, say dead whose semantics would be of always deadlocking, never affecting the constraint store).

By applying this operation to the above piece of example, we can eliminate ask(false) $\rightarrow$ stop, obtaining

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
   bidder_a(Bs, As) ||
   ask(true) → tell([LastBid|As] = [HisBid|HisList']) ||
        ...
```

Now we don't see any reason for not eliminating the guard ask(true) altogether. This can indeed be done via the following operation.

DEFINITION 3.7 (CONSERVATIVE ASK EVALUATION) *Consider the declaration*

$$\mathsf{d}: \ \mathsf{H} \leftarrow \mathsf{C}[\mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{B}]$$

We can transform d *into the declaration*

$$\mathsf{d}': \ \mathsf{H} \leftarrow \mathsf{C}[\mathsf{B}] \qquad \square$$

This operation, although trivial, is subject of debate. In fact, Sahlin in [19] defines a similar operation, with the crucial distinction that the choice might still have more than one branch, in other words, in the system of [19] one is allowed to simplify the agent $\mathsf{C}[\mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{A} + \mathsf{ask}(\mathsf{b}) \rightarrow \mathsf{B}]$ to the agent $\mathsf{C}[\mathsf{A}]$, even if b is satisfiable. Ultimately, one is allowed to replace the agent $\mathsf{C}[\mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{A} + \mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{B}]$ either with $\mathsf{C}[\mathsf{A}]$ or with $\mathsf{C}[\mathsf{B}]$, indifferently. Such an operation is clearly more widely applicable than the one we have presented (hence the attribute "conservative" in the definition above) but is bound to be *incomplete*, i.e. to lead to the loss of potentially successful branches. Nevertheless, Sahlin argues that an ask evaluation such as the one defined above is potentially too restrictive for a number of useful optimization. We agree with

the statement only partially, nevertheless, the system we propose will eventually be equipped with a non-conservative guard evaluation operation as well (which of course, if employed, will lead to weaker correctness results).

In our example program, the application of these branch elimination and conservative ask evaluation leads to the following:

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
    bidder_a(Bs, As) ||
    tell([LastBid|As] = [HisBid|HisList']) ||
    make_new_bid_b(HisBid,MyBid) ||
       ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits")
    + ask(MyBid ≠ quit) → tell(Bs = [MyBid|Bs']) ||
          tell(MyBid ≠ quit) ||
          bidder_b(HisList',Bs')
```

Via a tell evaluation of tell([LastBid|As] = [HisBid|HisList']), this simplifies to:

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
    bidder_a(Bs, As) ||
    make_new_bid_b(LastBid,MyBid) ||
       ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits")
    + ask(MyBid ≠ quit) → tell(Bs = [MyBid|Bs']) ||
          tell(MyBid ≠ quit) ||
          bidder_b(As,Bs')
```

3.4. Distribution

A crucial operation in our transformation system is the *distribution*, which consists of bringing an agent inside a choice as follows: from the agent

$$A \parallel \sum_i \mathsf{ask}(c_i) \rightarrow B_i,$$

we want to obtain the agent $\sum_i \mathsf{ask}(c_i) \rightarrow (A \parallel B_i)$. This operation was introduced for the first time in the context of CLP in [7], and requires delicate applicability conditions, as it can easily introduce deadlock situation: consider for instance the following contrived program D.

```
p(Y) ← q(X) || (ask(X >= 0) → tell(Y=0)
q(0) ← stop
```

In this program, the process D.p(Y) originates the derivation $\langle \mathsf{D.p(Y)}, \mathsf{true}\rangle \rightarrow^* \langle \mathsf{D.stop}, \mathsf{Y} = 0\rangle$. However, if we blindly apply the distribution operation to the first definition we would change D into:

```
p(Y) ← ask(X >= 0) → (q(X) || tell(Y=0))
```

and now we have that $\langle \mathsf{D.p(Y)}, \mathsf{true}\rangle$ generates only deadlocking derivations.

This situation is avoided by demanding that the agent being distributed will in any case not be able to produce any output before the choice is entered. This is done using the following notions of *required variable*. Recall that we denote by Stop any agent which contains only stop and || constructs.

DEFINITION 3.8 (REQUIRED VARIABLE) *Let* $\mathsf{D.A}$ *be a process. We say that* $\mathsf{D.A}$ requires *the variable* X *iff, for each satisfiable constraint* c *such that* $\mathcal{D} \models \exists_{\mathsf{X}} c \leftrightarrow c$, $\langle \mathsf{D.A}, c \rangle$ *has at least one finite derivation and moreover* $\langle \mathsf{D.A}, c \rangle \rightarrow^* \langle \mathsf{D.A}', c' \rangle$ *implies that* $\mathcal{D} \models \exists_{\_\tilde{z}} c \leftrightarrow \exists_{\_\tilde{z}} c'$, *where* $\tilde{z} = Var(\mathsf{A})$. □

In other words, the process $\mathsf{D.A}$ requires the variable X if, in the moment that the global store does not contain any information on X, then $\mathsf{D.A}$ cannot produce any information which affect the variables occurring in A and has at least one finite derivation. Even though the above notion is not decidable in general, in some cases it is easy to individuate required variables. For example it is immediate to see that, in our program, bidder_a(Bs, As) requires Bs: in fact the derivation starting in bidder_a(Bs, As) suspends (without having provided any output) after one step and resumes only when Bs has been instantiated. This example could be easily generalized. We can now give the formal definition of the distribution operation.

DEFINITION 3.9 (DISTRIBUTION) *Consider a declaration*

$$\mathsf{d}: \ \mathsf{H} \leftarrow \mathsf{C}[\mathsf{A} \parallel \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow \mathsf{B}_i]$$

The distribution *of* A *in* d *yields as result the definition*

$$\mathsf{d}': \ \mathsf{H} \leftarrow \mathsf{C}[\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow (\mathsf{A} \parallel \mathsf{B}_i)]$$

provided that A *requires a variable which does not occur in* H *nor in* C. □

The above applicability condition ensures that bringing A in the scope of the $\mathsf{ask}(c_i)$'s will not introduce deadlocking derivations: In fact it is intuitively clear that the fact that A requires a variable X implies, by definition, that A can produce some output only in the moment that X is instantiated, but since X does not occur in H nor in C, we have that this can only happen once the choice is entered. Summarizing, the applicability conditions ensure that (in the initial definition) A might produce an output only after the choice is entered. This ensures that A cannot have an influence on the choice itself, and can be thus safely brought inside.

In our example, since the agent bidder_a(Bs, As) requires the variable Bs, which occurs only inside the ask guards, we can safely apply the distributive operation. The result is the following program.

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_b(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits") ||
                                  bidder_a(Bs, As)
   + ask(MyBid ≠ quit) → tell(Bs = [MyBid|Bs']) ||
        tell(MyBid ≠ quit) ||
        bidder_a(Bs, As) ||
        bidder_b(As, Bs')
```

In this program we can now evaluate the construct tell(Bs = [MyBid|Bs']) obtaining (it is true that the variable Bs here occurs also elsewhere in the

definition, but since it occurs only on choice-branches different than the one on which the considered agent lies, we can assume it to be renamed):

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_b(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits") ||
                          bidder_a(Bs, As)
   + ask(MyBid ≠ quit) → tell(MyBid ≠ quit) ||
        bidder_a([MyBid|Bs'], As) ||
        bidder_b(As, Bs')
```

Before we introduce the fold operation, let us clean up the program a bit further: by properly transforming the agent bidder_a(Bs, As) in the first ask branch, we easily obtain:

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_b(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits") || stop
   + ask(MyBid ≠ quit) → tell(MyBid ≠ quit) ||
        bidder_a([MyBid|Bs'], As) ||
        bidder_b(As, Bs')
```

The just introduced stop agent can now safely be removed.

3.5. Folding

The folding operation has a special rôle in the panorama of the transformation operations. This is due to the fact that it allows to introduce recursion in a definition, often making it independent from the previous definitions. As previously mentioned, the applicability conditions that we use here for the folding operation do not depend on the transformation history: we only require that the declaration used to fold an agent appear in the initial program. We now need the following.

DEFINITION 3.10. *A* transformation sequence *is a sequence of programs*

$$\mathsf{D}_0, \ldots, \mathsf{D}_n,$$

in which D_0 *is an* initial program *and each* D_{i+1}*, is obtained from* D_i *via one of the following transformation operations: definition introduction, unfolding, distribution, guard simplification, branch elimination, conservative guard evaluation and folding.*

We also need the notion of *guarding context*. Intuitively, a context $\mathsf{C}[\,]$ is *guarding* if the "hole" appears in the scope of an ask guard[7]. Here $\equiv$ indicates syntactic equality.

DEFINITION 3.11 (GUARDING CONTEXT) *A context* $\mathsf{C}[\,]$ *is a* guarding context *iff*

$$\mathsf{C}[\,] \;\equiv\; \mathsf{C}'[\textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \to \mathsf{A}_i] \quad \text{and } \mathsf{A}_j = \mathsf{C}''[\,] \text{ for some } j \in [1,n]. \qquad \square$$

[7] The scope of the ask guard in $\mathsf{ask}(c) \to \mathsf{A}$ is A.

We can finally give the definition of folding:

DEFINITION 3.12 (FOLDING) *Let* $\mathsf{D}_0, \ldots, \mathsf{D}_i$, $i \geq 0$, *be a transformation sequence. Consider two definitions.*

$$\begin{array}{lll} \mathsf{d}: & \mathsf{H} \leftarrow \mathsf{C}[\mathsf{A}] & \in \mathsf{D}_i \\ \mathsf{f}: & \mathsf{B} \leftarrow \mathsf{A} & \in \mathsf{D}_0 \end{array}$$

If $\mathsf{C}[\,]$ *is a* guarding context *then* folding A *in* d *consists of replacing* d *by*

$$\mathsf{d}': \quad \mathsf{H} \leftarrow \mathsf{C}[\mathsf{B}] \quad \in \mathsf{D}_{i+1}$$

(it is assumed here that d *and* f *are suitably renamed so that the variables they have in common are only the ones occurring in* A*).* □

The reach of this operation is best shown via our example. We can now fold auction_left(MyBid) in the above definition, and obtain:

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_b(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits")
   + ask(MyBid ≠ quit) → auction_left(MyBid)
```

Now, by performing an identical optimization on auction_left, we can also obtain:

```
auction_left(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_a(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("a quits")
   + ask(MyBid ≠ quit) → auction_right(MyBid)
```

This part of the transformation shows in a striking way one of the main benefits of the folding operation: the saving of synchronization points. Notice that in the initial program the two bidders had to "wait" for each other. In principle they were working in parallel, but in practice they were always acting sequentially, since one always had to wait for the bid of the competitor. The transformation allowed us to discover this sequentiality and to obtain an equivalent program in which the sequentiality is exploited to eliminate all suspension points, which are known to be one of the major overhead sources. Furthermore, the transformation allows a drastic save of computational *space*. Notice that in the initial definition the parallel composition of the two bidders leads to the construction of two lists containing all the bids done so far. After the transformation we have a definition which does not build the list any longer, and which, by exploiting a straightforward optimization can employ only *constant* space.

4. CORRECTNESS

Any transformation system must be useful (i.e. allow useful transformations and optimization) and – most importantly – *correct*, i.e., it must guarantee that the resulting program is in some sense equivalent to the one we have started with. Having at hand a formal semantics for our paradigm, we defines *correctness* as follows.

DEFINITION 4.1 (CORRECTNESS) *A transformation sequence* $\mathsf{D}_0, \ldots, \mathsf{D}_n$ *is called*

- partially correct *iff for each agent* A *we have that* $\mathcal{O}(\mathsf{D}_0.\mathsf{A}) \supseteq \mathcal{O}(\mathsf{D}_n.\mathsf{A})$
- complete *iff for each agent* A *we have that* $\mathcal{O}(\mathsf{D}_0.\mathsf{A}) \subseteq \mathcal{O}(\mathsf{D}_n.\mathsf{A})$
- totally correct *iff it is both partially correct and complete.* □

So a transformation is *partially correct* iff nothing is added to the semantics of the initial program and is *complete* iff no semantic information is lost during the transformation. We can now state the main result of this paper.

THEOREM 4.2 (TOTAL CORRECTNESS) *Let* $\mathsf{D}_0, \ldots, \mathsf{D}_n$ *be a transformation sequence. Then* $\mathsf{D}_0, \ldots, \mathsf{D}_n$ *is* totally correct. □

This theorem is originally inspired by the one of Tamaki and Sato for pure logic programs [24], and has retained some of its notation. Of course the similarities don't go much further, as demonstrated by the fact that in our transformation system the applicability conditions of folding operation do not depend on the transformation history (while allowing the introduction of recursion), and that the folding definitions are allowed to be recursive (the distinction between P_{new} and P_{old} of [24] is now superfluous).

It is important to notice that – given the definition of observable we are adopting (Definition 2.1) – the initial program D_0 and the final one D_n have exactly the same successful derivation and the same deadlocked derivation. The first feature (regarding successful derivations) is to some extent the one we expect and require from a transformation, because it corresponds to the intuition that D_n "produces the same results" of D_0. Nevertheless, also the second feature (preservation of deadlock derivation) has an important rôle. Firstly, it ensures that the transformation does not introduce deadlock point, which is of crucial importance when we are using the transformation for optimizing a program. Secondly, this feature allows to use the transformation as a tool for proving deadlock freeness (i.e., absence of deadlock). In fact, if, after the transformation we can prove or or see that the process $\mathsf{D}_n.\mathsf{A}$ does never deadlock, then we are also sure that $\mathsf{D}_0.\mathsf{A}$ does not deadlock either.

5. RELATED WORK

In the literature, there exist three papers which are relatively closely related to the present one: de Francesco and Santone's [9], Ueda and Furukawa's [25], and Sahlin's [19]: in [9] it is presented a transformation system for CCS ([16]), in [25] it is defined a transformation system for Guarded Horn Clauses, while in [19] it is presented a transformation system for AKL.

Common to all three cases is that our proposal improves on them by introducing new operations such as the distribution, the techniques for the simplification of constraints, branch elimination and conservative guard evaluation

(though, some constraint simplification is done in [19] as well). Because of this, the transformation system we are proposing can be regarded as an extension of the ones in the paper above. Notice that without the above-mentioned operations the transformation of our example would not be possible. Further, we provide a more flexible definition for the folding operation, which allows the folding clause to be recursive, and frees the *initial program* from having to be partitioned in P_{new} and P_{old}.

Other minor differences between our paper and [25, 19] are the following ones. Compared to [25], our systems takes advantage of the greater flexibility of the CCP (wrt GHC). For instance, we can define the unfolding as a simple body replacement operation without any additional applicability condition, while this is not the case for GHC. As previously mentioned, differently from our case in [19] here it is considered a definition of *ask evaluation* which allows to remove potentially selectable branches; the consequence is that the resulting transformation system is only *partially* (thus not totally) correct. We should mention that in [19] two preliminary assumptions on the "scheduling" are made in such a way that this limitation is actually less constraining that it might appear. In any case, as we already said, the extended version of this transformation system will encompass an operation of *non-conservative* guard expansion, analogous to the one of [19] (and which – if employed – will necessarily lead to weaker correctness results).

Concluding, we want to mention that a previous work of the authors on the subject is [7] which focuses primarily on CLP paradigm (with dynamic scheduling), and is concerned with the preservation of deadlock derivation along a transformation. In [7], for the first time, it was employed a transformation system in order to prove absence of deadlock of a program (HAMMING). The second part of [7] contains a sketch of a primitive version of an unfold/fold transformation for CCP programs. Nevertheless, the system we are presenting here is (not only much more extended, but also) different in nature from [7]. This is clear if one compares the definitions of folding, which, it is worth reminding, is *the* central operation in an Unfold/Fold transformation system. In [7] this operation requires severe constraints on the initial program and applicability conditions which rely on the *transformation history*, while here the only requirement is that the folding has to take place inside a guarding context, which is a plain syntactic condition. As a consequence we have the following

– This system is – generally speaking – of much broader applicability.

All limitations on the initial programs are dropped. Ultimately, the folding definition is allowed to be recursive (which is really a step forward in the context of folding operations which are themselves capable of introducing recursion). Of course – being the two systems of different nature – one can invent an example transformation which is doable with the tools of [7] but not with the ones here presented. We strongly believe that such cases regard contrived examples of no practical relevances.

– The folding operation presented here is much simpler.

This is of relevance given the fact that the complexity of applicability of the folding operation has always been one of the major obstacle both in implementing it and in making it accessible to a wider audience.

In particular, as opposed to virtually all fold operations which enable to introduce recursion presented so far (the only exception being [9]), the applicability of the folding operation does not depend on the transformation history, (which has always been one of the "obscure sides" of it) but it relies on plain syntactic criteria.

We also should mention that because of the structural differences, the proofs for this paper are necessarily completely different.

Moreover, we have introduced new operations. In particular the guard simplification (which brings along the *branch elimination* and the *conservative guard evaluation*) is of crucial importance in order to have a transformation system which allows fruitful optimizations. Concluding, another fundamental operation for CCP – the distributive operation – has now simpler applicability conditions, which help in checking it in a much more straightforward way.

References

1. N. Bensaou and I. Guessarian. Transforming Constraint Logic Programs. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*, 1994.
2. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
3. K.L. Clark and S. Sickel. Predicate logic: a calculus for deriving programs. In *Proceedings of IJCAI'77*, pages 419–120, 1977.
4. F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 1998. to appear.
5. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166(1):101–146, 1996.
6. S. Etalle and M. Gabbrielli. Partial evaluation of concurrent constraint languages. *ACM Computing Surveys*, September 1998.
7. S. Etalle, M. Gabbrielli, and E. Marchiori. A Transformation System for CLP with Dynamic Scheduling and CCP. In *ACM–SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulation*. ACM Press, 1997.
8. S. Etalle, M. Gabbrielli, and M. C. Meo. Unfold/Fold Transformations of CCP Programs. In *Proc. 9th International Conference on Concurrency Theory*, pages 348–363. Springer-Verlag, 1998.
9. N. De Francesco and A. Santone. Unfold/fold transformation of concurrent processes. In H. Kuchen and S.Doaitse Swierstra, editors, *Proc. 8th Int'l Symp. on Programming Languages: Implementations, Logics and Programs*, volume 1140, pages 167–181. Springer-Verlag, 1996.

10. C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, April 1981.
11. N. Jørgensen, K. Marriot, and S. Michaylov. Some Global Compile-Time Optimizations for CLP($\mathcal{R}$). In *Proc. 1991 Int'l Symposium on Logic Programming*, pages 420–434, 1991.
12. T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 413–422. Institute for New Generation Computer Technology, Tokyo, 1988.
13. H. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Proc. Ninth ACM Symposium on Principles of Programming Languages*, pages 255–267. ACM, 1982.
14. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.
15. M.J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, March 1993.
16. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
17. T Mogensen and P Sestoft. Partial evaluation. In A. Kent and J.G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. M. Dekker, 1997.
18. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
19. D. Sahlin. Partial Evaluation of AKL. In *Proceedings of the First International Conference on Concurrent Constraint Programming*, 1995.
20. V. A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989.
21. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York, 1990.
22. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 1991.
23. G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, number 1000 in LNCS. Springer-Verlag, 1995. see www.ps.uni-sb.de/oz/.
24. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.
25. K. Ueda and K. Furukawa. Transformation rules for GHC Programs. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 582–591. Institute for New Generation Computer Technology, Tokyo, 1988.

A. Appendix: Sketch of the Proofs

In this section we prove that our transformation system is *totally correct*. For space reasons, proof are sketched. In what follows, we refer to a fixed *transformation sequence* $\mathsf{D}_0, \ldots, \mathsf{D}_n$. We start with the following result, concerning partial correctness.

Proposition A.1 (Partial correctness) *Let* $\mathsf{i} \in [1, \mathsf{n}-1]$. *If, for each agent* A, $\mathcal{O}(\mathsf{D}_0.\mathsf{A}) = \mathcal{O}(\mathsf{D}_\mathsf{i}.\mathsf{A})$ *then, for each agent* A, $\mathcal{O}(\mathsf{D}_\mathsf{i}.\mathsf{A}) \supseteq \mathcal{O}(\mathsf{D}_{\mathsf{i}+1}.\mathsf{A})$.

Proof. *By induction on the length of the derivations.* □

Definition A.2 (Weight) *Let* ξ *be a derivation. We denote by* $wh(\xi)$ *the number of derivation steps in* ξ *which use rule R2. Given an agent* A *and a pair of satisfiable constraints* c, d, *we then define the* success weight $\mathsf{w_s}(\mathsf{A}, \mathsf{c}, \mathsf{d})$ *of the agent* A *wrt the constraints* c *and* d *as follows*

$$\mathsf{w_s}(\mathsf{A}, \mathsf{c}, \mathsf{d}) = min\{n \mid \begin{array}{l} n = wh(\xi) \text{ and } \xi \text{ is a derivation} \\ \langle \mathsf{D}_0.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D}_0.\mathsf{Stop}, \mathsf{d}'\rangle \not\rightarrow \\ \text{with } \exists\_\mathsf{Var}(\mathsf{A},\mathsf{c})\mathsf{d}' = \exists\_\mathsf{Var}(\mathsf{A},\mathsf{c})\mathsf{d}\}. \end{array}$$

Note that, according to this definition, the success weight is computed by considering successful derivations. The notion of weight, as well as the following one of *descent derivation*, can be analogously defined for deadlocked derivations as well, by simply replacing in the definition the agent Stop with a generic agent $\mathsf{B} \not\equiv \mathsf{Stop}$. Here – to keep the notation to a minimum – we sketch the part of the demonstration relative to the success derivation; for this reason the above weight is the only one we need.

In the total correctness proof we also make use of the concept of *descent derivations*. Intuitively, these are derivations which can be split into two parts: the first one, up to the first ask evaluation, is performed in the program D_i while the second one is carried out in D_0.

Definition A.3 (Descent derivation) *Let* D_i *and* D_0 *be two programs. We call a derivation in* $\mathsf{D}_\mathsf{i} \cup \mathsf{D}_0$ *a (successful)* descent derivation *if it has the form*

$$\langle \mathsf{D}_\mathsf{i}.\mathsf{A}_1, \mathsf{c}_1\rangle \rightarrow^* \langle \mathsf{D}_\mathsf{i}.\mathsf{A}_\mathsf{m}, \mathsf{c}_\mathsf{m}\rangle \rightarrow \langle \mathsf{D}_0.\mathsf{A}_{\mathsf{m}+1}, \mathsf{c}_{\mathsf{m}+1}\rangle \rightarrow^* \langle \mathsf{D}_0.\mathsf{Stop}, \mathsf{c}_\mathsf{n}\rangle \not\rightarrow$$

where $m \in [1, n]$[8] *and the following conditions hold:*

(a) the first $m-1$ *derivation steps do not use rule R2;*

(b) the m-th derivation step $\langle \mathsf{D}_\mathsf{i}.\mathsf{A}_\mathsf{m}, \mathsf{c}_\mathsf{m}\rangle \rightarrow \langle \mathsf{D}_0.\mathsf{A}_{\mathsf{m}+1}, \mathsf{c}_{\mathsf{m}+1}\rangle$ *uses rule R2;*

(c) $\mathsf{w_s}(\mathsf{A}_1, \mathsf{c}_1, \mathsf{c}_\mathsf{n}) > \mathsf{w_s}(\mathsf{A}_{\mathsf{m}+1}, \mathsf{c}_{\mathsf{m}+1}, \mathsf{c}_\mathsf{n})$. □

This definition is inspired by the definition of *descent clause* of [12]; however, here we use a different notion of weight and different conditions on them.

We need one final concept.

[8] If $m = n$ we can write indifferently $\langle \mathsf{D}_\mathsf{i}.\mathsf{Stop}, \mathsf{c}_\mathsf{n}\rangle$ or $\langle \mathsf{D}_0.\mathsf{Stop}, \mathsf{c}_\mathsf{n}\rangle$ to denote the last configuration of the derivation.

DEFINITION A.4. *We call the program* $\mathsf{D_i}$ weight complete *iff, for any agent* A *and pair of constraints* c, d, *the following hold: if there exists a derivation*

$$\langle \mathsf{D_0}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_0}.\mathsf{B}, \mathsf{d}\rangle \not\rightarrow$$

then there exists a descent derivation

$$\langle \mathsf{D_i}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_0}.\mathsf{B}', \mathsf{d}'\rangle \not\rightarrow$$

where $\exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}' = \exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}$ *and* $\mathsf{B} \equiv \mathsf{Stop}$ *iff* $\mathsf{B}' \equiv \mathsf{Stop}$. □

So $\mathsf{D_i}$ is weight complete if we can reconstruct the semantics of $\mathsf{D_0}$ by using only (successful and deadlocked) descent derivations in $\mathsf{D_i} \cup \mathsf{D_0}$. We now show that if $\mathsf{D_i}$ is weight complete then no new observables lost during the transformation (i.e., that the transformation is *totally correct*). This is the content of the following.

PROPOSITION A.5. *If* $\mathsf{D_i}$ *is weight complete then, for any agent* A, $\mathcal{O}(\mathsf{D_0}.\mathsf{A}) \subseteq \mathcal{O}(\mathsf{D_i}.\mathsf{A})$.

PROOF. Recall that consider now only the case of successful derivations: the one of deadlocked derivations is analogous and omitted. Assume that there exists a (finite, successful) derivation $\langle \mathsf{D_0}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_0}.\mathsf{Stop}, \mathsf{d}\rangle$. We show, by induction on the weight of $(\mathsf{A}, \mathsf{c}, \mathsf{d})$, that there exists a derivation $\langle \mathsf{D_i}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_i}.\mathsf{Stop}, \mathsf{d}'\rangle$, where $\exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}' = \exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}$.

Base Case. If $\mathsf{w_s}(\mathsf{A}, \mathsf{c}, \mathsf{d}) = 0$ then, since $\mathsf{D_i}$ is weight complete, from Definition A.3 it follows that there exists a descent derivation in $\mathsf{D_i} \cup \mathsf{D_0}$ of the form $\langle \mathsf{D_i}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_i}.\mathsf{Stop}, \mathsf{d}'\rangle$ where $\exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}' = \exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}$, rule $R2$ is not used and therefore each derivation step is done in $\mathsf{D_i}$.

Inductive Case. Assume that $\mathsf{w_s}(\mathsf{A}, \mathsf{c}, \mathsf{d}) = \mathsf{n}$. Since $\mathsf{D_i}$ is weight complete there exists a descent derivation in $\mathsf{D_i} \cup \mathsf{D_0}$

$$\xi : \langle \mathsf{D_i}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_0}.\mathsf{Stop}, \mathsf{d}'\rangle,$$

where $\exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}' = \exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}$. If rule $R2$ is not used in ξ then the proof is the same as in the previous case. Otherwise ξ has the form

$$\langle \mathsf{D_i}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_i}.\mathsf{A_m}, \mathsf{c_m}\rangle \rightarrow \langle \mathsf{D_0}.\mathsf{A_{m+1}}, \mathsf{c_{m+1}}\rangle \rightarrow^* \langle \mathsf{D_0}.\mathsf{Stop}, \mathsf{d}'\rangle$$

where $\mathsf{w_s}(\mathsf{A}, \mathsf{c}, \mathsf{d}') > \mathsf{w_s}(\mathsf{A_{m+1}}, \mathsf{c_{m+1}}, \mathsf{d}')$. Let $\xi' : \langle \mathsf{D_i}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_i}.\mathsf{A_m}, \mathsf{c_m}\rangle \rightarrow \langle \mathsf{D_i}.\mathsf{A_{m+1}}, \mathsf{c_{m+1}}\rangle$. By inductive hypothesis, there exists a derivation

$$\xi'' : \langle \mathsf{D_i}.\mathsf{A_{m+1}}, \mathsf{c_{m+1}}\rangle \rightarrow^* \langle \mathsf{D_i}.\mathsf{Stop}, \mathsf{d}''\rangle$$

where $\exists_{-\mathsf{Var}(\mathsf{A_{m+1}},\mathsf{c_{m+1}})}\mathsf{d}'' = \exists_{-\mathsf{Var}(\mathsf{A_{m+1}},\mathsf{c_{m+1}})}\mathsf{d}'$. Without loss of generality, we can assume that $\mathsf{Var}(\xi') \cap \mathsf{Var}(\xi'') = \mathsf{Var}(\mathsf{A_{m+1}}, \mathsf{c_{m+1}})$ and hence there exists a derivation $\langle \mathsf{D_i}.\mathsf{A}, \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D_i}.\mathsf{Stop}, \mathsf{d}''\rangle$. Finally by our hypothesis on variables, $\exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}'' = \exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}(\mathsf{c_{m+1}} \wedge \exists_{-\mathsf{Var}(\mathsf{A_{m+1}},\mathsf{c_{m+1}})}\mathsf{d}'') =$

$\exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}(\mathsf{c}_{m+1} \wedge \exists_{-\mathsf{Var}(\mathsf{A}_{m+1},\mathsf{c}_{m+1})}\mathsf{d}') = \exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}' = \exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}$, which concludes the proof. □

We can now prove our main theorem, let us state it again.

Theorem 4.2 (Total Correctness) Let $\mathsf{D}_0, \ldots, \mathsf{D}_n$ be a transformation sequence. Then

– $\mathsf{D}_0, \ldots, \mathsf{D}_n$ is *totally correct*.

PROOF. (Sketch; again, recall that we are considering only successful derivations, and that the case of deadlocking ones is analogous). The proof proceeds by showing simultaneously, by induction on i, that for $i \in [0, n]$:

1. for any pair of constraints c, d and context $\mathsf{C}[\ ]$, if $\mathsf{def} : \mathsf{p}(\tilde{\mathsf{t}}) \leftarrow \mathsf{B}$ is a declaration in D_i then there exists a constraint d' such that $\mathsf{w_s}(\mathsf{C}[\mathsf{B} \parallel \mathsf{tell}(\tilde{\mathsf{s}} = \tilde{\mathsf{t}})], \mathsf{c}, \mathsf{d}') \leq \mathsf{w_s}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{s}})], \mathsf{c}, \mathsf{d})$ and $\exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{s}})])}\mathsf{d} = \exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{s}})])}\mathsf{d}'$.

   Moreover, if $i > 0$, for any pair of constraints c, d and context $\mathsf{C}[\ ]$, if $\mathsf{p}(\tilde{\mathsf{t}}) \leftarrow \mathsf{B}$ is a declaration in $\mathsf{D}_{\mathsf{i}-1}$ and $\mathsf{p}(\tilde{\mathsf{t}}) \leftarrow \mathsf{B}'$ is in D_i, then there exists a constraint d' such that $\mathsf{w_s}(\mathsf{C}[\mathsf{B}'], \mathsf{c}, \mathsf{d}') \leq \mathsf{w_s}(\mathsf{C}[\mathsf{B}], \mathsf{c}, \mathsf{d})$ and $\exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})])}\mathsf{d} = \exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})])}\mathsf{d}'$ (and analogously for the deadlock weights);
2. $\mathcal{O}(\mathsf{D}_0) = \mathcal{O}(\mathsf{D}_\mathsf{i})$;
3. D_i is weight complete.

Base case. We just need to prove that D_0 is weight complete. Assume that $\langle \mathsf{c}, \mathsf{d}, \mathsf{ss} \rangle \in \mathcal{O}(\mathsf{D}_0.\mathsf{A})$. Then there exists a derivation

$$\xi : \ \langle \mathsf{D}_0.\mathsf{A}, \mathsf{c} \rangle \rightarrow^* \langle \mathsf{D}_0.\mathsf{Stop}, \mathsf{d}' \rangle \not\rightarrow .$$

whose weight is minimal and where $\mathsf{d} = \exists_{-\mathsf{Var}(\mathsf{A},\mathsf{c})}\mathsf{d}'$. It follows from Definition A.3 that ξ is a descent derivation.

Induction step. Assume that the thesis holds for $i - 1 \geq 0$. The proof of 1. is done by considering various cases, according to the transformation performed when moving from $\mathsf{D}_{\mathsf{i}-1}$ to D_i. We show here only the case in which this operation is a *backward instantiation* (the other cases are similar or simpler). Moreover, we consider the case of successful derivations only, as the case of deadlocked ones is analogous.

Assume that the operation employed for obtaining D_i from $\mathsf{D}_{\mathsf{i}-1}$ is a backward instantiation. Let:

– $\mathsf{def} : \mathsf{p}(\tilde{\mathsf{t}}) \leftarrow \mathsf{C}'[\mathsf{q}(\tilde{\mathsf{r}})]$ and $\mathsf{b} : \mathsf{q}(\tilde{\mathsf{v}}) \leftarrow \mathsf{tell}(\mathsf{e}) \parallel \mathsf{B}$ be declarations in $\mathsf{D}_{\mathsf{i}-1}$,
– e' be constraint such that $\mathcal{D} \models \mathsf{e} \rightarrow \mathsf{e}'$ and
– $\mathsf{def}' : \mathsf{p}(\tilde{\mathsf{t}}) \leftarrow \mathsf{C}'[\mathsf{q}(\tilde{\mathsf{r}}) \parallel \mathsf{tell}(\mathsf{e}') \parallel \mathsf{tell}(\tilde{\mathsf{r}} = \tilde{\mathsf{v}})]$ is the result of the transformation (in D_i).

We prove that for any $\mathsf{C}[\,]$, constraints c and d, there exists a constraint d' such that

$$\mathsf{w_s}(\mathsf{C}[\mathsf{q}(\tilde{\mathsf{r}}) \parallel \mathsf{tell}(\mathsf{e}') \parallel \mathsf{tell}(\tilde{\mathsf{r}} = \tilde{\mathsf{v}})], \mathsf{c}, \mathsf{d}') \leq \mathsf{w_s}(\mathsf{C}[\mathsf{q}(\tilde{\mathsf{r}})], \mathsf{c}, \mathsf{d})$$

and $\exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})])}\mathsf{d} = \exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})])}\mathsf{d}'$. Then the thesis follows by inductive hypothesis.

Notice that, since we are considering only successful derivations, by definition of $\mathsf{D}_{\mathsf{i}-1}$, we have to consider only derivation

$$\langle \mathsf{D}_{\mathsf{i}-1}.\mathsf{C}[\mathsf{q}(\tilde{\mathsf{r}})], \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D}_{\mathsf{i}-1}.\mathsf{Stop}, \mathsf{f}\rangle \not\rightarrow$$

such that $\mathcal{D} \models \mathsf{f} \rightarrow \mathsf{e} \wedge \tilde{\mathsf{r}} = \tilde{\mathsf{v}}$ (for the proof for the deadlock case we also have to consider the situation in which the above implication does not hold). Then since by definition $\mathcal{D} \models \mathsf{e} \rightarrow \mathsf{e}'$ and by inductive hypothesis $\mathcal{O}(\mathsf{D}_0) = \mathcal{O}(\mathsf{D}_{\mathsf{i}-1})$, we have that for any derivation

$$\xi = \langle \mathsf{D}_0.\mathsf{C}[\mathsf{q}(\tilde{\mathsf{r}})], \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D}_0.\mathsf{Stop}, \mathsf{d}\rangle \not\rightarrow,$$

$\mathcal{D} \models \mathsf{d} \rightarrow \mathsf{e}' \wedge \tilde{\mathsf{r}} = \tilde{\mathsf{v}}$ holds. Then there exists a derivation

$$\xi' = \langle \mathsf{D}_0.\mathsf{C}[\mathsf{q}(\tilde{\mathsf{r}}) \parallel \mathsf{tell}(\mathsf{e}') \parallel \mathsf{tell}(\tilde{\mathsf{r}} = \tilde{\mathsf{v}})], \mathsf{c}\rangle \rightarrow^* \langle \mathsf{D}_0.\mathsf{Stop}, \mathsf{d}'\rangle \not\rightarrow,$$

which performs exactly the same steps of the derivation ξ plus two tell actions and such that $\exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})])}\mathsf{d} = \exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})])}\mathsf{d}'$. Therefore we have $\mathsf{wh}(\xi) = \mathsf{wh}(\xi')$ and, by definition of $\mathsf{w_s}$, that $\mathsf{w_s}(\mathsf{C}[\mathsf{q}(\tilde{\mathsf{r}})], \mathsf{c}, \mathsf{d}) \geq \mathsf{w_s}(\mathsf{C}[\mathsf{q}(\tilde{\mathsf{r}}) \parallel \mathsf{tell}(\mathsf{e}') \parallel \mathsf{tell}(\tilde{\mathsf{r}} = \tilde{\mathsf{v}})], \mathsf{c}, \exists_{-\mathsf{Var}(\mathsf{C}[\mathsf{p}(\tilde{\mathsf{t}})])}\mathsf{d}')$.

In order to prove 2. and 3. observe that, by induction hypothesis, we have that $\mathcal{O}(\mathsf{D}_0) = \mathcal{O}(\mathsf{D}_{\mathsf{i}-1})$, and that $\mathsf{D}_{\mathsf{i}-1}$ is weight complete. From Propositions A.1 and A.5 it follows that if D_i is weight complete then $\mathcal{O}(\mathsf{D}_0) = \mathcal{O}(\mathsf{D}_\mathsf{i})$. So we simply have to prove that D_i is weight complete. This follows easily by the definition of descent derivation by using point 1. □