PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Dependent Types and Formal Synthesis

F. K. Hanna and N. Daeche

Phil. Trans. R. Soc. Lond. A 1992 **339**, 121-135
doi: 10.1098/rsta.1992.0029

Dependent types and formal synthesis

By F. K. Hanna and N. Daeche

Electronic Engineering Laboratories, University of Kent, Canterbury CT2 7NT, U.K.

The relative advantages offered by the use of dependent types (rather than polymorphic ones) in a higher-order logic used for reasoning about digital systems are explored. Dependent types and subtypes are shown to provide an effective means of expressing the bounded, parametrized types typically encountered in this field. Heuristic methods can be used to minimize problems arising from the loss of decidable type-checking.

A second topic discussed is formal synthesis, an approach to design in which the activities of behavioural synthesis and of formal verification are combined. The starting point is a behavioural specification, the end result is a specification of an implementation together with a proof of its correctness.

1. Introduction

Typed higher-order logic has been shown to be, from many points of view, a most effective formalism for describing and formally reasoning about digital systems (Gordon 1986; Hanna & Daeche 1986; Melham 1987; Joyce 1988). One area where some difficulty is experienced, however, is in the inability of ordinary type structures (that is, those built on the ordinary cartesian product $\times$ and function space $\rightarrow$ operators) to capture with precision many of the bounded parametrized types informally used by the digital designer. Another area where difficulties are experienced is in managing, even with computational assistance, the sheer complexity of formally verifying realistically sized systems. In this paper we address these two problems in turn. To overcome the first we propose the use of *dependent* types, a natural generalization of ordinary types. To alleviate the second, we propose an approach named *formal synthesis* in which the activities of behavioural synthesis and of formal verification are combined.

2. Dependent types

The digital designer works at a level of abstraction close to the physical world. As a result, many of the signals dealt with are bounded (since each one is an abstraction of a finite number of distinguishable physical states) or are parametrized (since silicon real estate is a scarce resource and hence word lengths tend to be tailored to specific requirements). Consider for example the types needed to describe the signals at the ports of a *multiplexer*; the 'output' port carries a signal of type *bit* (a bounded subrange of the natural numbers), the 'select' port carries one of type *n-bit word* and the 'input' port carries one of type 2^n*-bit word*. It is important that a formalism for describing and reasoning about digital systems is able to express and manipulate

Phil. Trans. R. Soc. Lond. A (1992) **339**, 121–135
Printed in Great Britain

types like these fluently. We have found that the so-called *dependent* types (that feature in Martin-Löf's intuitionistic type theory (Martin-Löf 1984)) fill this role admirably; they are soundly based in mathematical logic and yet are natural and intuitive to use. Veritas is a *design logic* (that is, a logic specifically intended for supporting formal design methods) that incorporates such types. In the following sections we will be illustrating the properties and uses of dependent types by developing (in Veritas) some aspects of number theory and some behavioural specifications for typical digital devices.

The main features of Veritas, briefly stated, are the following.

1. It is a typed, higher-order logic (Enderton 1972). Its rules of inference are classical (i.e. not intuitionistic) and its form is a sequent calculus.

2. It provides the following kinds of types. (i) Primitive types: the type *bool* of propositional truth values (*true* and *false*) and a series of *type universes* $U0$, $U1, \ldots$ (in effect, types of types). (ii) Recursive types ('datatypes'), on which functions may be defined by primitive recursion. (iii) Subtypes. (iv) Dependent Σ and Π types (these are a natural generalization of ordinary $\times$ and $\rightarrow$ types).

3. Terms do not have unique types; rather, particular term-type combinations ('judgements', in Martin-Löf's terminology) are inferred to be well formed.

4. Types are themselves terms and thus the ordinary term constructors (application, abstraction, etc.) may be used equally on types as on ordinary terms.

(A fuller discussion of the design rationale of the logic may be found in Hanna *et al.* (1990).)

The logic has been computationally implemented by embedding it as a collection of abstract types in a programming language along the general lines of Edinburgh LCF (Gordon *et al.* 1979). The notation used throughout this paper is the Veritas-90 version of the logic and is identical to that which the computational implementation of the logic both accepts and generates.

(*a*) *Datatypes and primitive recursion*

We begin our description of the type structure of Veritas by considering datatypes and primitive recursion. *Datatype declarations* are used to introduce new types together with their *constructors*. For example, the declaration:

datatype *colour* = *red* | *green* | *blue*

introduces a type *colour* having exactly three elements.

The natural numbers 0, 1, 2, ..., may be introduced by a *recursive* datatype:

operator {′ **post**} **datatype** *nat* = 0 | *nat*′.

This introduces the type *nat* together with its two constructors 0 and prime (here defined as a *post*fixed operator). The latter constructor is known as the *successor* function and is of type *nat* → *nat*. Using these constructors, the first few natural numbers may be defined:

$$1 \hat{=} 0'; \quad 2 \hat{=} 1'; \quad 3 \hat{=} 2'.$$

Functions on datatypes are defined by primitive recursion (PR). Since, when written directly in their low-level form, PR terms have a somewhat rebarbative appearance, the language provides an alternative high-level form in which they may

be expressed. For example, a typical PR function definition on *nat* may be written in high-level form as

define *even* **by**
even $0 = true$ |
even $n' = \neg(even\ n)$
end.

(Read as: *even* is defined as the function that maps 0 to *true* and that maps n' to the complement ('$\neg$') of the value of *even* n.) We note, however, that this high level form is just an alternative presentation of the actual low-level PR form, which in this case is

$even \hat{=}$ **recurse** $true\,|\,(\lambda n{:}nat.\,\lambda b{:}bool.\,\neg b)$ **end**.

The usual arithmetic operators on *nat* are easily defined; we give a few examples. The addition function is

define $\{+\}$ **by** $n+0 = n\,|\,n+m' = (n+m)'$ **end**.

The predecessor function (the left inverse of the successor function) is

define *pred* **by** *pred* $0 = 0\,|\,pred\ n' = n$ **end**.

The 'proper subtraction' function ($\dot{-}$) and the comparison predicate ($\leqslant$) are:

define $\{\dot{-}\}$ **by** $n \dot{-} 0 = n\,|\,n \dot{-} m' = pred\ (n \dot{-} m)$ **end**;

operator $\{\leqslant\}$ **in** $n \leqslant m \hat{=} (n \dot{-} m) = 0$.

Notice that all functions defined in the language are, by construction, total (that is, they are defined for all values of their arguments). This indicates, therefore, that it would be impossible to construct a definition for a function like division on the natural numbers since division by 0 is undefined.

(*b*) *Subtypes*

A *subtype* of a type is defined by giving the characteristic predicate that specifies membership of the subtype. For instance, the type nat^+ comprising the positive natural numbers may be defined by

$nat^+ \hat{=} \{n{:}nat\,|\,n > 0\}$

Whereas with a datatype it is immediately apparent which terms belong to it, with a subtype it is necessary to infer membership by showing that the term satisfies the characteristic predicate of the subtype. Consider for instance the term $2+2$; by construction, it is a member of the type *nat*. However, by demonstrating the theorem $\vdash (2+2) > 0$ it may be inferred that it is also a member of the type nat^+.

A typical application for the type nat^+ is in specifying a type for the division function $div{:}nat \to nat^+ \to nat$. By constraining its second argument to range only over non-zero numbers, the function can be well defined over its entire domain.

Since types are terms, type constructing functions taking ordinary terms as their arguments may be defined. For example, consider the function N defined by

$N\ n \hat{=} \{m{:}nat\,|\,m < n\}$.

This function takes a number n and yields the subtype of *nat* containing numbers less than n. For instance, the type $N\,3$ comprises the numbers 0, 1 and 2.

The type constructor N nicely illustrates two features of the logic. Firstly, since it

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

is itself a term, it has its own type; this is $N : nat \to U0$ (recollect that $U0$ is the type of (small) types). Secondly, it shows that types may be empty as, for instance, is the type $N\,0$ – in ordinary typed predicate calculus, all types are deemed to be non-empty.

(c) *Dependent types*

Consider the function *mod* (modulo) and in particular, what type it ought to be given. It *could* be given the same type as the function *div* (above) was given; this would guarantee that it was well defined over its entire domain. However, it could with advantage be given the *dependent* type:

$$mod : nat \to [n : nat^+] \to N\ n.$$

(Read as: *mod* takes an argument of type *nat* and then an argument n of type nat^+ and it yields a result of type $N\ n$.) A type like this is called 'dependent' since the type of the range of the function *depends* upon the value of its arguments. For example, the type of the term $mod\ m\ 3$ is $N\,3$. In many contexts information like this is almost as useful as the value itself of a term.

The parameter n in the above dependent type is a bound variable. An alternative way of writing dependent types is to use a prefix binder notation. For example, a dependent type like $[n : nat^+] \to N\ n$ may be written as $\Pi n : nat^+.\ N\ n$ (where 'Π' is the binder). Such types are thus often called Π-types. It is easy to see that ordinary function ('$\to$') types are simply a special case of Π-types in which the binding is vacuous. For instance, the type $nat \to bool$ may equally be written as $\Pi n : nat.\ bool$.

Just as $\to$-types may be generalized to Π-types, so also may $\times$-types (cartesian product types) be generalized to Σ-types. For instance, the type $[n : nat] \times N\ n$ (alternatively written $\Sigma n : nat.\ N\ n$) comprises all pairs whose first component n is of type *nat* and whose second component is of type $N\ n$. For example, a pair like $(2, 0)$ belongs to this type whereas one like $(2, 3)$ does not.

As a significant example of dependent types and subtypes, consider a primitive recursive definition for the functions *div* and *mod* discussed above. Since these two functions are closely related, it is convenient to merge them into a single function (named '//', as a binary operator) of type:

$$\{//\} : nat \to [n : nat^+] \to nat \times N\ n$$

such that the value of $a\,//\,b$ is the pair $(div\ a\ b)$, $(mod\ a\ b)$. Here is the definition:

define $\{//\} : nat \to [n : nat^+] \to nat \times N\ n$ **by**

$0\,//\,n = (0, 0)\,|$

$m'\,//\,n =$ **let** $(q, r) = m\,//\,n$ **in**

if $r' = n$ **then** $(q', 0)$ **else** (q, r')

end

(Read as: '//' is a binary operator (of the specified dependent type) defined (by primitive recursion) such that the value of $0\,//\,0$ is $(0, 0)$, and the value of $m'\,//\,n$ is defined in terms of the pair (q, r) (the quotient and remainder of $m\,//\,n$) ***either*** as (q, r') ***or*** (if this would overflow) as $(q'\,0)$.)

This is, in fact, quite a subtle definition and it repays detailed study. For example, to take just one point, consider what would happen if the function had instead been given the less restrictive type

$$nat \to [n : nat] \to nat \times N\ n$$

(i.e. so that division by zero was not excluded). In that case it would not be possible to establish that the term $(0, 0)$ in the definition was of type $nat \times N\ n$ (as it is required to be) since if n took on the value 0 (which it would no longer be excluded from doing) the type $N\ n$ would be empty. Thus the overall definition would fail to be well typed.

3. Specifications of digital systems

We now move on to discuss the application of the dependent types, subtypes and datatypes of Veritas to the specification of digital systems. It is our aim to show that such specifications can be both very precise and yet intuitively natural.

Digital systems, for the most part, operate on bits and on fixed length sequences of bits termed words. The natural choice for a type to represent bits is $bit \mathrel{\hat{=}} N\ 2$. For representing n-bit words, however, several choices are possible. One way is as a mapping from $N\ n$ (the index type) to *bit*. For this, a type-constructing function W may be defined:

$$W\ n \mathrel{\hat{=}} (N\ n) \to bit$$

so that words of length m are represented by the type $W\ m$. Using W, a type *byte* for representing 8-bit words (bytes) may be defined as $byte \mathrel{\hat{=}} W\ 8$. Then, if $B : byte$ is a particular byte, its individual bits may be accessed by functional application, thus: $B\ 7$, $B\ 6, \ldots, B\ 0$. Following conventional mathematical practice, Veritas treats subscripting as denoting application and so the same terms may also be written as $B_7, B_6, \ldots, B_0$. References outside the index range $N\ 8$ are, of course, impossible even to express since such terms (for example, B_8) are not well typed.

A very common operation on words is selecting subwords. The following function

$$select : [n : nat] \to W\ n \to [i : N\ n] \to [j : N(n \dot{-} i)' \to W\ j]$$

defined by

$$select\ w\ i\ j\ k \mathrel{\hat{=}} w\ (i + k)$$

carries out this operation. It takes a word-length n, a word w of that length, an index i into that word, a subword length j and it yields a word of length j. For instance, the term $select_8\ B\ 4\ 2$, of type $W\ 2$, describes the two-bit subword comprising bits B_4 and B_5 of the original word. Notice carefully how dependent subtypes are used to constrain the allowable ranges of the various arguments; the type discipline of the language prevents badly defined subwords from being expressed.

(*a*) *Binary numerals*

In many cases, words are used to represent binary numerals. The valuation function, *val*, for such numerals is defined by:

define $val : [n : nat] \to W\ n \to N(2 \uparrow n)$ **by**

$val\ 0\ w = 0 \mid$

$val\ m'\ w = (2 \uparrow m) \times w_m + val\ m\ w$

end

(Read as: The function *val* takes a number n (the word length), then an n-bit word and yields a result of type $N\ 2^n$. The value of a zero-length word is zero; the value of an $(m+1)$-bit word is 2^m times the values of its m.s.b. plus the value of the remaining m-bit word.)

There are two interesting features about this definition. Firstly, notice how the type of *val* explicitly expresses the relation between word length and value – information often relied upon by the designer when intuitively justifying the correctness of a design. Secondly, notice that *val* is an isomorphism; two n-bit numerals are equal if and only if their values are equal; that is

$$\forall n{:}nat;\ A, B{:}W\ n.$$
$$(A = B) = (val_n\ A = val_n\ B).$$

This useful relation arises because the indexing type of numerals is limited to a subrange of *nat*. Had the function W instead been defined as $W\ n \mathrel{\hat{=}} nat \to bit$, then equality of numbers would not entail equality of their corresponding n-bit numerals.

(*b*) *Alternative representation for words*

Although from many points of view the representation of binary numerals given above is very effective, it does suffer from the shortcoming that defining numeric literals is awkward. For instance, a definition of the 4-bit binary numeral for twelve (i.e. 1100_2) in this representation would be:

$$TWELVE \mathrel{\hat{=}} \lambda i{:}N\ 4.\ \textbf{if}\ i = 2 \vee i = 3\quad \textbf{then}\ 1\ \textbf{else}\ 0.$$

An alternative representation that overcomes this problem is instead to represent binary numerals as tuples (formed by iterated pairing). For this, first introduce a singleton datatype **datatype** *binary* = *Bin* (that will be used to represent zero-length tuples) and then define a type-constructing function T as

define $T{:}nat \to U0$ **by** $T\ 0 = binary \mid T\ n' = bit \times T\ n$ **end**.

With this definition, a type like, for example, $T\ 4$, is equal to

$$bit \times bit \times bit \times bit \times binary$$

and numeric literals of this type are easily written, as, for example,

$$TWELVE \mathrel{\hat{=}} 1,\ 1,\ 0,\ 0,\ Bin.$$

This definition of the parametrized type T is, in effect, a simulation of fixed-length lists, with the pairing operator (comma) playing the role of *cons* and the constant *Bin* playing the role of *nil*. Using binary numerals defined by T, the valuation function *val* still retains its dependent type but is now defined as

define $val{:}[n{:}nat] \to T\ n \to N(2 \uparrow n)$ **by**
$val\ 0\ w = 0 \mid$
$val\ m'\ (b, u) = (2 \uparrow m) \times b + val\ m\ u$
end.

A straightforward generalization of the function T allows the base of a numeral to be parametrized as well so that a numeric literal to any positive base may easily be expressed. For instance, it allows the number 1992 to be written as 1, 9, 9, 2, $Base_{10}$ of type $T_{(10,4)}$.

In some digital systems, numbers are represented in a hybrid code called *binary coded decimal* (BCD). A representation for such numerals is easily defined. First introduce a type to represent BCD digits:

$$digit \mathrel{\hat{=}} \{d{:}T\ 4 \mid val_4\ d < 10\}$$

together with the definitions for the ten digits $zero, \ldots, nine$:

$$zero \hat{=} 0,\ 0,\ 0,\ 0,\ Bin; \quad \ldots \quad nine \hat{=} 1,\ 0,\ 0,\ 1,\ Bin$$

Then (following a similar scheme to that used for the definition of T) introduce:

datatype $bcd = BCD$;

define $TT : nat \to U0$ **by**

$TT\ 0 = bcd \mid$

$TT\ n' = digit \times TT\ n$

end.

A typical BCD-encoded numeral, for instance 1992, is then expressible as *one*, *nine*, *nine*, *two*, *BCD* of type TT_4.

A closely related version of the parametrized type TT may be defined for representing (as a tuple of tuples) an n by m array of bits; a typical use for such a type is for defining the contents of a read-only memory.

(*c*) *Specifications of digital systems*

We now move on to illustrate the role that dependent types can play in specifying the behaviour or the structure of digital systems. Throughout, we will be using the convention that predicates are used to describe specifications of digital devices or systems. The value (or *extension*) of the predicate will describe a device's (steady-state) behaviour by characterizing the allowable combinations of signals that can occur at its ports. In addition, the form (or *intension*) of the predicate will sometimes be used to indicate the structure of the device.

We begin with a simple example, that of a *multiplexer*, a device that is used to select one signal from one of its 2^n input signals. Its behaviour may be specified by the parametrized predicate:

$$multiplexer : [n : nat] \to W\ n \times W\ (2 \uparrow n) \times bit \to bool$$

defined by

$$multiplexer_n\ (sel,\ inputs,\ output) \hat{=} output = inputs\ (val_n\ sel)$$

Here, *sel* represents the n-bit selector word, *inputs* represents the 2^n input signals and *output* represents the output signal. The type and behavioural predicate for a particular sized multiplexer are obtained by application of the function. For example, the type of the specification for a 16-way multiplexer is

$$multiplexer_4 : W\ 4 \times W\ 16 \times bit \to bool.$$

(*d*) *Binary addition*

A wide range of arithmetic operations may be implemented by one-dimensional iteratively structured digital systems. We take as a simple example the ordinary parallel binary adder and consider specifications of its behaviour at different levels of abstraction.

First consider the specification of an adder viewed at the abstract level. Notice that, even at this level, the device is seen, not as performing addition on the (infinite) type *nat*, but rather as performing addition modulo m on the bounded type $N\ m$ (for some positive m). We assume that the adder (see figure 1*a*) has two input ports (a and

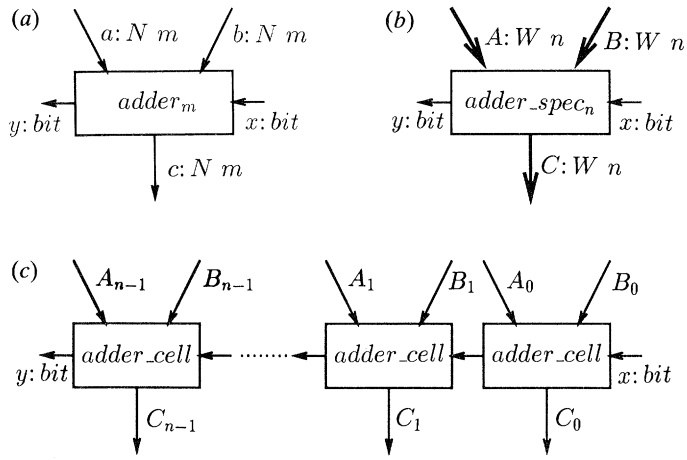

Figure 1. (*a*) Addition viewed at the abstract level. (*b*) Addition viewed at the binary-numeral level of abstraction. (*c*) Implementation of a binary adder.

b), an output port (*c*) and carry-in and carry-out ports (*x* and *y*). Its desired behaviour is described by the parametrized predicate

$$adder\colon [m\colon nat^{+}] \to N\ \ m \times N\ \ m \times N\ \ m \times bit \times bit \to bool$$

defined by

$$adder_m\ (a,\ b,\ c,\ x,\ y) \mathrel{\hat{=}} (y,\ c) = (a + b + x)//m$$

(that is, the main output c is the sum modulo m of the inputs, and the carry output is the sum divided by m).

Next consider the specification of the adder seen at the 'binary numeral' level of abstraction (figure 1*b*). Now the type of its operands is parametrized by the word-length n and the function val_n is used as the abstraction function. Its desired behaviour is described by the parametrized predicate

$$adder\_spec\colon [n\colon nat] \to W\ \ n \times W\ \ n \times W\ \ n \times bit \times bit \to bool$$

defined by

$$adder\_spec_n\ (A,\ B,\ C,\ x,\ y) \mathrel{\hat{=}} adder_{(2\uparrow n)}\ (val_n\ A,\ val_n\ B,\ val_n\ C,\ x,\ y).$$

This specification may be realized (figure 1*c*) by an iterative structure consisting of n adder-cells serially composed. The behaviour of this structure is described by the predicate

$$adder\_impl\colon [n\colon nat] \to W\ \ n \times W\ \ n \times W\ \ n \times bit \times bit \to bool$$

defined by

$$adder\_impl_n\ (A,\ B,\ C,\ x,\ y) \mathrel{\hat{=}}$$
$$\exists X\colon W\ \ n'.$$
$$(X_0 = x) \wedge (X_n = y) \wedge$$
$$\forall i\colon N\ \ n.\ adder\_cell\ (A_i,\ B_i,\ C_i,\ X_i,\ X_{i'}).$$

(The $\exists$-quantified variable X represents the internal $(n+1)$-length word of carry signals and the $\forall$-quantified variable i is used to index over the words A, B, C and X.)

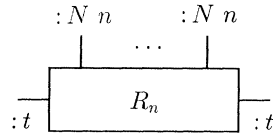

Figure 2. Specification R_n of a device having an arbitrary number of inputs and outputs, each of type $N\ n$, and a carry-in and carry-out each of type t.

The full-adder cell used in the above definition is described by the predicate

$$adder\_cell : bit \times bit \times bit \times bit \times bit \rightarrow bool$$

nicely defined by

$$adder\_cell\ (a, b, c, x, y) \hat{=} (y, c) = (a+b+x)//2.$$

Finally, the criterion expressing the correctness of the implementation relative to the specification may be expressed as

$$\forall n : nat;\ A, B, C : W\ n;\ x, y : bit.$$
$$adder\_impl_n\ (A, B, C, x, y) \Rightarrow adder\_spec_n\ (A, B, C, x, y)$$

or, very much more concisely, as $\forall n : nat.\ adder\_impl_n \sqsupseteq adder\_spec_n$, if the operator $\sqsupseteq$ ('is at least as strong as') has been defined.

One rather interesting aspect of the above set of definitions is that the predicate for the behaviour of an individual adder-cell can be expressed in terms of that for the overall adder. The relation between the two predicates is simply $adder\_cell = adder_2$.

An alternative way of describing this correspondence is to say that a realization of the (abstract) specification $adder_{(2\uparrow n)}$ may (for any value of n) be obtained by composing n instances of the specification $adder_2$ and viewing the result through the abstraction function val_n. In fact, it is not difficult to see that this result generalizes from base-2 representation to base-b ones. That is, for any word-length n and any positive base b, the specification $adder_{(b\uparrow n)}$ may be realized by composing n instances of the adder-cell specification $adder\_cell_b$ and interpreting the operands as numerals to base-b.

(*e*) *Factorization theorem*

The obvious question posed by the above observation is whether it can be generalized still further. To explore this, consider generalizing the *adder* relation to an arbitrary parametrized relation R that takes as its arguments a number n and then some arbitrary (but fixed) number of operands of type $N\ n$ (each representing an input or an output signal) and two operands of an arbitrary type t (representing a generalized 'carry' signal), thus

$$R : [n : nat] \rightarrow N\ n \times \ldots \times N\ n \times t \times t \rightarrow bool.$$

The question (see figure 2) is then whether, given a particular relation R, the abstract specification $R_{(m\uparrow n)}$ can be realized at a concrete level (using length-n, base-m numerals) by the serial composition of n instances of the relation R_m, for any n and any positive m. If it can be, then we say that R is *iteratively implementable*.

On inspection, many common arithmetic operations are found to be iteratively implementable, including: addition and subtraction (and, as special cases, increment and decrement), comparison and subrange inclusion, multiplication by a constant, division by a constant and remainder modulo a constant. However, not all operations

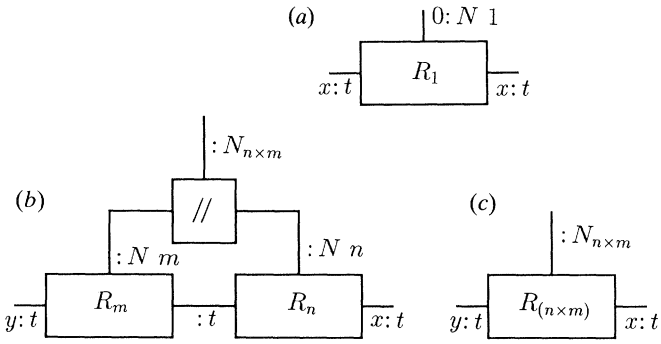

Figure 3. A parametrized specification R is said to be *proper* if R_1 behaves as shown in (a). It is said to be *factorizable* if (for arbitrary $n, m : nat^+$) the composed relations R_m and R_n shown in (b) behave equivalently to the relation $R_{(n\times m)}$ shown in (c).

share this property; for example, squaring does not. We have, however, found two simple properties (termed *proper* and *factorizable*) and have established:

Factorization theorem. *A parametrized relation which is both proper and factorizable is iteratively implementable.*

A diagrammatic definition of these two properties is shown in figure 3. A formal statement of the theorem and notes on the computational checking of its proof may be found in Hanna *et al.* (1989*a*).

The factorization theorem is useful from two points of view. Firstly, it offers a general design method for synthesizing iterative implementations of arithmetic operations. Secondly, it greatly simplifies the task of formally verifying the correctness of such an implementation by dividing the task into two separate components. On the one hand is a proof of the factorization theorem itself; because this is independent of the (possibly intricate) details of any particular relation R it is relatively simple to establish. Further, because it is a general result, it only ever has to be proven once. Then, on the other hand, is a proof that the particular relation R being considered is proper and factorizable. Because this does not involve any aspects of the iterative numerical representation (for example, binary-coded decimal numerals), it too is relatively simple.

(*f*) *Example*

As a simple illustration of the application of the factorization theorem, consider the design of an iterative structure to extract the remainder modulo m of a number. The abstract specification of the operation (on numbers in the subrange $N\ n$) is described by the parametrized predicate

$$modulo : [m : nat^+] \times [n : nat] \rightarrow N\ \ n \times N\ \ m \times N\ \ m \rightarrow bool$$

defined by

$$modulo_{(m,\,n)}(a, x, y) \mathrel{\hat{=}} x = mod\,(n \times y + a)\ \ m.$$

This relation can be shown to be both proper and factorizable, and hence can be asserted to be iteratively implementable. Now suppose that an implementation of this operation is required for determining remainder modulo 3 on binary numerals. Given the above result, it can straightaway be asserted (see figure 4) that an

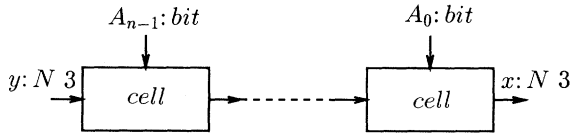

Figure 4. Implementation of the operation *remainder modulo 3* for binary numerals.

implementation can be realized by serially composing k instances of the relation *cell* of type

$$cell: bit \times N\ 3 \times N\ 3 \rightarrow bool$$

defined by $cell \mathrel{\hat{=}} modulo\ 3\ 2$ or, equivalently stated:

$$cell\ (a, x, y) \mathrel{\hat{=}} x = mod\ (2 \times y + a)\ 3$$

and it can be asserted that the resultant structure will (for numerals of length k) satisfy the relation $modulo_{(3,\,2\uparrow k)}$ when viewed through the abstraction function val_k.

(*g*) *Discussion*

The type system examined in this paper is derived from the dependent types of Martin-Löf's ITT. The natural alternative to this kind of type system is one based on Milner polymorphism. In this section, we explore the relative merits of these two type systems in the context of a design logic intended for reasoning about digital hardware.

In polymorphic type systems, type expressions may contain type variables and thus represent families of types. Every well-typed term possesses a unique most general type (called its principal type) and the type-checking problem is decidable. By contrast, with a dependent type system, terms do not possess unique types and the type-checking problem is undecidable.

If a design logic is being used as no more than a semi-formal notation for writing specifications then whether or not type checking is decidable is not a matter of any consequence. With either kind of system, the well-typing of terms is almost always evident by inspection. However, if the terms and theorems of a design logic are to be subjected to computational checking – as almost all serious applications of verification technology demand – then the distinction is a significant one. On the one hand, the user of a polymorphically typed logic need have no concern with type-checking (since it can be dealt with algorithmically) and is free to concentrate on the central activity of theorem proving. On the other hand, the user of a dependently typed logic is forced to deal explicitly with the type-checking problem. Typically this involves coercing the types of terms by rewriting their types with equivalent types or by restricting their types to subtypes. An example will illustrate this. Suppose that the typed term $1:bit$ is required (for instance, as an argument for the predicate *adder_cell* defined earlier). By construction, the term 1 has the type *nat*. To show that it can be coerced to have the type *bit* involves the following pattern of inferences:

$$\dfrac{\dfrac{\dfrac{1:nat \quad \vdash 1 < 2}{1:\{n:nat \mid n < 2\}} \quad \vdash N\ \ n = \{m:nat \mid m < n\}}{1:N\ 2} \quad \vdash bit = N\ 2}{1:bit}$$

(that is, using the theorem $\vdash 1 < 2$ the term $1:nat$ is first injected into the subtype $\{n:nat \mid n < 2\}$ and then this subtype is rewritten, first as $N\ 2$ and then as *bit*).

The difference in the amount of labour required to guide computational theorem proving with either polymorphic or dependent types turns out, however, not to be as great as the above example may suggest, for the following two reasons.

1. Most of the deduction carried out during type-checking with dependent types tends to be required at some stage or other when using either type system. For example, if a term like *div m n* occurs in a proof, it is likely that, somewhere, the assertion $\vdash n > 0$ will be required. With dependent types, it will be required at the start for type-checking (to establish the typing $n : nat^+$), whereas with polymorphic types it will almost certainly be required later on, in order to discharge a hypothesis $n > 0$ occurring in the definition of *div*.

2. Just as the routine aspects of deduction can be partly automated by using goal-directed methods and domain-specific heuristics ('tactics'), so also can the routine aspects of type-checking be partly automated. For example, tactics can be written to encapsulate the patterns of inference (such as that shown above) required to type check terms whose types involve either of the type-constructing functions N and W that have figured extensively in this paper.

Given that the loss of fully decidable type-checking is the price paid for the use of dependent types, what compensating advantages do such types confer? We can summarize these advantages in three broad groups.

Firstly, dependent types and subtypes allow types to be formulated for a given specification that encompass *exactly* the sets of values required. This allows the bounded types (for instance, *bit*) characteristic of digital hardware to be represented directly in their natural form. Further, it allows functions (for example, *div*) to be defined that are both total on their domain and yet do not contain redundant or arbitrary information. In turn, this leads to specifications with useful mathematical properties (for example, numerals being isomorphic to numbers).

Secondly, dependent types provide a more structured (and thus higher level) notation, by virtue of both the value *and* the type of a term carrying useful information. For example, the type alone of the term *mod n m* carries the information that its value is bounded by the subtype $N\ m$. In addition (a point not explored in this paper), dependent product types provide an effective means of defining abstract data types (for instance, integer or bounded stack).

Thirdly, dependent types allow the types of specifications to be parametrized not only with types (as do polymorphic type systems) but also with values. This allows generic specifications to be defined (for example, those involving n-digit numerals to base b). Not only do such generic specifications provide an economical way of covering many specific instances but, by virtue of having abstracted away from the irrelevant detail of particular instances, they can often lead to extra insight. For example, the factorization theorem discussed earlier arose naturally from a generalization of this kind.

4. Formal synthesis

The sheer labour involved in carrying through computationally checked formal verification (irrespective of the particular species of logic used) is arguably the only factor limiting its widespread industrial application. Designs that can readily be seen to be correct by inspection can often take days or weeks of a skilled designer's time to computationally verify.

There are two reasons that may be discerned for this immense disparity. One is due

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

to the difference (the 'semantic gap') between the conceptual, high level at which the human designer reasons as compared with the very low level at which the primitive rules of inference of a design logic are framed. The use of goal-directed theorem proving supported by extensive sets of high-level, domain-specific tactics can go some way towards narrowing this gap. Additionally, the enhanced expressiveness that dependent types allow can contribute to higher-level specifications and to more generic patterns of inference.

The second reason for the disparity is due to the lack of effective sharing of information between the activities of design and of verification. While some aspects (for example, the choice of representations for abstract types internal to a design) of an implementation may be essentially arbitrary, for others the designer will have well-defined reasons for believing that they imply the correctness of the implementation. Either way, if design and verification are carried out separately, then the verifier will have to rediscover reasons for believing the design to be correct and then, laboriously, communicate these to the verification system. It is these observations that have motivated the development of an approach, named *formal synthesis*, in which the activities of behavioural synthesis and formal verification are combined.

(*a*) *Approach*

The formal synthesis approach (Hanna *et al.* 1989*b*) is closely based on the goal-directed methods introduced in the Edinburgh LCF theorem-prover (Gordon *et al.* 1979). The process is a recursive one; the designer starts at the top level with a behavioural specification ϕ of the desired system and with the two-fold goal of determining a specification ψ for a realizable implementation together with a proof $\vdash \psi \sqsupseteq \phi$ of its correctness. To *achieve* this goal, the designer recursively selects from a set of functions known as *techniques*; these play an analogous role to those of tactics in theorem proving. Each technique embodies a particular design step and comprises two parts: one (the subgoaling function) that generates a set of subgoals from the original goal and the other (the validating function) that takes a corresponding set of *achieved* subgoals and uses them to achieve the original goal. The subgoals that are generated may be either behavioural specifications (to which technique will later be applied to realize them) or propositions (to which tactics will later be applied to establish them as theorems).

(*b*) *Techniques*

The techniques used for formal synthesis are of many differing kinds. Basic, however, to any application is a set of general purpose techniques that embody simple design steps such as: splitting a specification containing a conjunction into two separate specifications, rewriting or strengthening a specification (either by using an existing theorem or by generating an appropriate theorem subgoal), or stripping an existential quantifier from a specification by introducing an internal signal. Techniques like these are analogous to the general purpose tactics (such as *generalize* or *rewrite*) used in theorem proving.

Many specifications involve abstract types which must, during the design process, be replaced by more concrete types; we term this process *elaboration*, the inverse of abstraction. There are three contexts in which elaboration occurs.

1. There is elaboration of an abstract type that is local (i.e. internal) to a specification. Here, the technique can select an arbitrary representation; for instance, it may represent the 2^n elements of an enumerated set by successive n-bit

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

binary numerals. The fact that the particular representation and abstraction function chosen by the subgoaling function of the technique is directly available to the validating function greatly reduces the need for further interaction with the designer.

2. There is elaboration of an abstract type that is global to the specification but its representation is free to be decided. This case is similar to the first one except that the theorem that is returned by the validating function now contains the chosen representation and abstraction function. An example of this is where the instruction set of a microprocessor that is to be designed is only specified abstractly (for instance, as **datatype** *instr* = *nop* | *lda* | *addr* | ...) and the designer is free to choose the binary codes for the instruction set.

3. There is elaboration of an abstract type that is global to the specification but its representation is defined in advance. An instance of this is in designing a microprocessor that is to be binary-code compatible with an existing model.

Many aspects of digital design can be defined algorithmically and hence are natural candidates for techniques. There are two approaches that may be adopted, corresponding to *opaque* or *transparent* techniques. With opaque techniques, any synthesis method may be used to generate a design and then, by quite independent methods, it correctness is established. A simple instance of this is where an exhaustive search algorithm is used to determine an optimal NAND-gate combinational circuit which is then verified by case analysis. By contrast, with transparent techniques, the synthesis algorithm that is used is lifted so that, as each step is undertaken, it carries out the corresponding validation. A simple instance of this is where a combinational circuit is being optimized by using propositional tautologies to eliminate redundant components.

(*c*) *Design trees*

Although it is possible for a designer to carry out formal synthesis by directly accessing techniques and tactics (by programming at the meta language level) it is, in practice, desirable to provide a high-level user interface, a goal-directed editor. Interestingly, because of the close similarity of formal synthesis to goal-directed theorem proving, such an editor function can be written polymorphically and specialized to either task by providing (as an argument to the function) either a library of techniques or one of tactics.

The act of the designer selecting techniques (and thus generating sets of subgoals) induces the generation of a tree, termed the *design tree*, whose arcs are labelled with design or theorem goals, and whose nodes are labelled with techniques or tactics. The designer has considerable freedom in deciding in which order to create the nodes of this tree. For instance, they may be created in a *prudent* order (proofs first, then designs), in an *exploratory* order (designs first, then proofs), or even in a *reckless* order (designs but no proofs). In a practical situation, where different aspects of a design tend to be more or less critical, or more or less obviously correct, this choice of order is a valuable freedom to have. The design tree itself when complete provides a permanent record, not only of a completed design, but also of the process of its creation and a proof of its correctness.

It is a pleasure to acknowledge the contribution of Dr Mark Longley to the development of the formal synthesis method described here, and of Dr Gareth Howells to the development of a reference implementation of Veritas-90 in the functional programming language Haskell.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

The SML implementation of the Veritas-90 theorem prover was carried out with sponsorship from International Computers Ltd, from Program Validation Ltd and from the U.K. Science and Engineering Research Council under Grant GR/F/3668.

References

Claesen, L. J. M. 1989 *Applied formal methods for correct VLSI design*. North-Holland.

Enderton, H. B. 1972 *A mathematical introduction to logic*. Academic Press.

Gordon, M., Milner, R. & Wadsworth, C. 1979 *Edinburgh LCF*. Lecture Notes in Computer Science, vol. 78, Springer-Verlag.

Gordon, M. J. 1986 Why higher-order logic is a good formalism for specifying and verifying hardware. In Milne & Subrahmanyam (1986), pp. 153–177.

Hanna, F. K. & Daeche, N. 1986 Specification and verification using higher-order logic; a case study. In Milne & Subrahmanyam (1986), pp. 179–213.

Hanna, F. K., Daeche, N. & Longley, M. 1989*a* Veritas^{+}: a specification language based on type theory. In Leeser & Brown (1989), pp. 358–379.

Hanna, F. K., Longley, M. & Daeche, N. 1989*b* Formal synthesis of digital systems. In Claesen (1989), pp. 153–169.

Hanna, F. K., Daeche, N. & Longley, M. 1990 Specification and verification using dependent types. *IEEE Trans. SE* **16**, 949–964.

Joyce, J. J. 1988 Formal specification and verification of microprocessor systems. *Tech. Rep.* no. 147. Computer Laboratory, University of Cambridge.

Leeser, M. & Brown, G. (eds) 1989 *Hardware specification, verification and synthesis: mathematical aspects*. Mathematical Sciences Institute Workshop, Cornell University, Ithaca, New York. Springer LNCS 408.

Martin-Löf, P. 1984 Constructive mathematics and computer programming. *Phil. Trans. R. Soc. Lond.* A **312**, 501–518.

Melham, T. F. 1987 Abstraction mechanisms for hardware verification. *Tech. Rep.* no. 106. Computer Laboratory, University of Cambridge.

Milne, G. J. & Subrahmanyam, P. (eds) 1986 *Formal aspects of VLSI design*. Amsterdam: North-Holland.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES