T. Melham, 'Using Recursive Types to Reason about Hardware
in Higher Order Logic', Technical Report No. 135, University of
Cambridge Computer Laboratory (May 1988).

Using Recursive Types to Reason about Hardware in Higher Order Logic

Thomas F. Melham

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

Abstract: *The expressive power of higher order logic makes it possible to define a wide variety of data types within the logic and to prove theorems that state the properties of these types concisely and abstractly. This paper describes how such defined data types can be used to support formal reasoning in higher order logic about the behaviour of hardware designs.*

First printed: May 1988
Reprinted with revisions: April 1990

An earlier version of this paper appears in: *The Fusion of Hardware Design and Verification*, ed. G.J. Milne (North-Holland, 1988), pp. 27–50.

Contents

Introduction

The aim of this paper is to show how *recursive data types* can be used to support formal reasoning in higher order logic about the behaviour of hardware devices. Two examples are given: the correctness proof of a class of tree-structured circuits, and the formulation and proof of assertions describing the relationship between two simple transistor models. In both examples, recursive types are used to state propositions about hardware in a general and concise way.

The organization of the paper is as follows. In Section 1, a brief review is given of the conventional techniques for specifying and verifying hardware in higher order logic. Section 2 discusses how recursive types can be added to higher order logic without making *ad hoc* extensions to the axioms of the logic. Sections 3 and 4 give two examples to show how such recursive types can be used to extend the techniques outlined in Section 1. Recursive types are used in both examples to model the structure of circuits independently of their behaviour.

1 Hardware Verification using Higher Order Logic

The basic techniques for specifying and proving the correctness of hardware using higher order logic are well established and are documented in several recent papers [1, 6, 9, 12]. To make this paper self-contained, a brief review is given in this section of these techniques. The version of higher order logic used is based on Church's type theory [3], extended with the type discipline of the LCF logic PPλ [7]. This formulation of higher order logic was developed by Mike Gordon for the HOL theorem prover [5] and is described in detail in [4].

1.1 Notation

The formulation of higher order logic used in this paper includes terms that correspond to the conventional notation of predicate calculus. A term of the form $P\,x$ expresses the proposition that x has the property P, and a term of the form $R(x, y)$ means that the relation R holds between x and y. The usual logical operators $\neg$, $\wedge$, $\vee$, $\supset$ and $\equiv$ denote negation, conjunction, disjunction, implication, and equivalence respectively. The universal and existential quantifiers $\forall$ and $\exists$ express the concepts of *every* and *some*: $\forall x. P\,x$ means that P holds for every value of x, and $\exists x. P\,x$ means that P holds for some (i.e. at least one) value of x. The additional quantifier $\exists!$ denotes unique existence: $\exists! x. P\,x$ means that P holds for exactly one value of x. Nested quantifiers of the form $\forall v_1.\, \forall v_2.\, \cdots \forall v_n.\, tm$ can also be written $\forall v_1\, v_2\, \cdots\, v_n.\, tm$. Other notation includes $(c \Rightarrow t_1 \,|\, t_2)$ to denote the conditional 'if c then t_1 else t_2', and $f \circ g$ to denote the composition of the functions f and g. The constants T and F denote the truth values *true* and *false*.

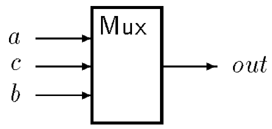
Higher order logic extends the conventional notation of predicate calculus in three significant ways: (1) variables are allowed to range over functions and predicates, (2) functions can take functions as arguments and return functions as results, and (3) functions can be written in the notation of the λ-calculus.[1] For example, the idea that a clock ck rises at some time t can be expressed in higher order logic as follows:

$$\forall ck\, t.\, (\mathsf{Rise}\; ck)\; t = \neg ck(t) \wedge ck(t{+}1)$$

In this definition, ck is a higher order variable ranging over functions from time (modelled by natural numbers) to booleans, and Rise is a higher order function that takes a function modelling a clock as an argument and yields a predicate on natural numbers as a result. Conventional practice is that function application in higher order logic associates to the left. For example, the term $(\mathsf{Rise}\; ck)\; t$ can also be written $\mathsf{Rise}\; ck\; t$.

1.2 Specifying Hardware Behaviour

The behaviour of hardware devices can be specified in higher order logic by defining predicates that state which combinations of values can appear on their external ports. Consider, for instance, the one-bit multiplexer shown below:



The behaviour of this device can be specified in logic by a four-place predicate Mux, defined such that the term '$\mathsf{Mux}(c, a, b, out)$' is true exactly when the

[1] λ-calculus notation will not, however, be used in this paper.

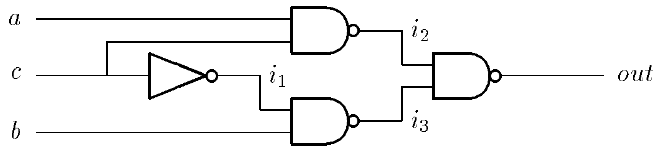
combination of the values of the variables c, a, b, and out is one that could occur on the corresponding ports of the device. The definition of Mux is:

$$\mathsf{Mux}(c, a, b, out) \equiv (out = (c \Rightarrow a \mid b))$$

In this specification the variables c, a, b, and out range over boolean truth-values. The predicate Mux asserts that the relationship between these values corresponds to the way a multiplexer works in practice: when the control line c is true, the value on the output out is equal to the value on the input a; and when c is false the value on out is equal to the value on b.

1.3 Specifying Hardware Structure

The behaviour of devices constructed by wiring together smaller devices can be represented in logic by conjoining the predicates that specify the behaviours of their components with the logical connective '$\wedge$' and using the existential quantifier '$\exists$' to hide internal signals [1, 6]. Consider, for example, the multiplexer implementation shown below:



If Inv and Nand are predicates that specify the behaviour of an inverter and a NAND-gate respectively, then this multiplexer implementation can be specified in logic by the predicate $\mathsf{Mux\_imp}$ defined as follows:

$$\begin{array}{l} \mathsf{Mux\_imp}(c, a, b, out) \equiv \\ \quad \exists i_1\, i_2\, i_3.\, \mathsf{Inv}(c, i_1) \wedge \mathsf{Nand}(a, c, i_2) \;\wedge \mathsf{Nand}(i_1, b, i_3) \wedge \mathsf{Nand}(i_2, i_3, out) \end{array}$$

In this definition the three internal wires i_1, i_2, and i_3 are 'hidden' from the external environment using the existential quantifier '$\exists$'. The definition of $\mathsf{Mux\_imp}$ states that the values which can appear on the external ports of the multiplexer are precisely those which satisfy the constraints imposed by the predicates modelling the four gates from which it is built.

1.4 Formulating Correctness

The predicate Mux defined in Section 1.2 specifies the intended behaviour of a one-bit multiplexer. The multiplexer circuit defined by $\mathsf{Mux\_imp}$ can be proved correct with respect to this specification by proving the following theorem:

$$\vdash \forall c\, a\, b\, out.\, \mathsf{Mux\_imp}(c, a, b, out) \equiv \mathsf{Mux}(c, a, b, out)$$

This theorem states that the values which can appear on the external ports of the multiplexer implementation are exactly those allowed by the specification of intended behaviour. The implementation defined by $\mathsf{Mux\_imp}$ is therefore correct with respect to the specification given by Mux.

In this simple example, the implementation predicate $\mathsf{Mux\_imp}$ is logically equivalent to the specification of intended behaviour Mux. For more complex circuits, however, it may be inappropriate to formulate correctness as logical equivalence. The behavioural specification that a large or complex circuit is expected to satisfy will typically be an *abstract* description of its intended behaviour. It may be only a partial specification; or it may be given in terms of higher-level data types or a different time-scale than the predicate defining the implementation. In this case, the correctness of an implementation Imp with respect to a specification Spec will not be a logical equivalence,

$$\vdash \forall i\, o.\, \mathsf{Imp}(i, o) \equiv \mathsf{Spec}(i, o)$$

but an implication of the form:

$$\vdash \forall i\, o.\, \mathsf{Imp}(i, o) \supset \mathsf{Spec}(\mathsf{Abs}(i, o))$$

where Abs is an abstraction function that maps input and output signals of the implementation description Imp to corresponding signals of the abstract specification Spec. For a discussion of various ways in which the correctness of circuits can be formulated using such abstraction functions see [12].

2 Recursive Types in Higher Order Logic

Higher order logic is a *typed* logic; every syntactically well-formed term of the logic must have a type that is consistent with the types of its subterms. Informally, types can be thought of as denoting sets of values, and terms can be thought of as denoting elements of these sets. The basic types of the version of higher order logic used in this paper include *bool* (denoting the set of boolean truth-values) and *num* (denoting the set of natural numbers). These two types are examples of type *constants*; they denote fixed sets of values. Types can be built from other types using type *operators*. An example is the primitive type operator '$\rightarrow$', denoting the function space operation on types. If ty_1 and ty_2 are types, then the type $ty_1 \rightarrow ty_2$ denotes the set of all total functions from values of type ty_1 to values of type ty_2. The syntax of types also includes type *variables*. These are written α, β, γ, etc., and are used to stand for 'any type'. Type variables occur in Church's formulation of higher order logic as *metavariables* ranging over types; in the version of higher order logic used here they are part of the object language.

Writing '$tm{:}ty$' indicates explicitly that the term tm has logical type ty. The function Rise defined in Section 1.1, for example, can be written with explicit type information as shown below:

$$\mathsf{Rise}{:}(num \rightarrow bool) \rightarrow (num \rightarrow bool)$$

Such type information will usually be omitted, however, when it is clear from the form or context of a term what its type must be.

As a syntactic device, types are necessary to avoid inconsistency. Without types, the expressive power gained by allowing variables to range over functions makes it possible to write paradoxical expressions in the logic that make it inconsistent (e.g. Russell's paradox). Ensuring that every term has a type which is consistent with those of its subterms makes such expressions syntactically ill-formed, and thus eliminates them from the logic.

The type expressions needed to prevent inconsistency have a very simple and economical syntax. All that is needed are the type constants *num* and *bool*, and types of the form $ty_1 \rightarrow ty_2$. In principle, every type needed for doing proofs in higher order logic can be written using only these primitive types. But in practice it is desirable to extend the syntax of types to include more kinds of types than are strictly necessary to prevent inconsistency.

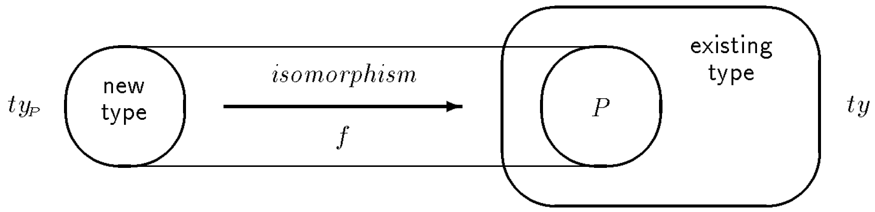
Extending the syntax of types in higher order logic allows types to play a *mathematical* role in reasoning about hardware, in addition to their purely *logical* role of eliminating inconsistency. In mathematical (and programming) practice the notion of types is used to make distinctions between variables that range over different kinds of values (e.g. numbers, pairs, lists, recursive structures, etc.). From this point of view, a type is the name of a commonly-used set of values of a particular kind, having certain well-defined properties. Such types are often characterized by sets of axioms that define their properties abstractly and concisely.

Many of the sorts of values that arise naturally in reasoning about hardware (e.g. bit-vectors) can be represented by types of this kind. As will be discussed in Section 3, devices whose components exhibit some form of recursively regular structure (e.g. adders and trees of gates) can also be represented by axiomatized data types. While the primitive types of the logic are in principle sufficient to represent these data structures, adding new types to the logic make it possible to formulate propositions about hardware in a more natural and concise way. This pragmatic motivation for a rich syntax of types is similar to the motivation for the use of abstract data types in high-level programming languages.

Section 2.1 below contains a brief explanation of how the syntax of types in higher order logic can be consistently extended using *type definitions*. These are analogous to abstract type definitions in programming languages like ML [10]; they define a new type by representing it by a set of values of an already existing type. Section 2.3 describes a class of *recursive* types which can be added to the logic using such type definitions.

2.1 Type Definitions

The type definition mechanism described in this section is based on a suggestion by Mike Fourman, which has been formalized by Mike Gordon in [4]. The idea is that a type definition is made by adding an axiom to the logic which asserts that a new type is isomorphic to an appropriate 'subset' of an existing type:



Suppose, for example, that ty is a type of the logic and $P{:}ty{\rightarrow}bool$ is a predicate on values of type ty that defines some useful subset of the set denoted by ty. A type definition introduces a new type constant ty_P which denotes a set having exactly the same properties as the subset defined by P. This is done by extending the syntax of types to include the new type constant ty_P

and then adding an axiom to the logic asserting that the set of values denoted by the new type is isomorphic to the set specified by P:

$$\vdash \exists f{:}ty_P{\to}ty.\ (\forall a_1\, a_2.\, f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, P\ r = (\exists a.\, r = f\ a)) \tag{1}$$

This axiom states that there is a function f from the new type ty_P to the existing type ty which is one-to-one and onto the subset defined by P. The function f can be thought of as a representation function that maps a value of the new type ty_P to the value of type ty that represents it. Because f is an isomorphism, it can be shown that the set denoted by ty_P has the same properties as the subset of ty defined by P. By adding this axiom to the logic, the new type ty_P is therefore defined in terms of the existing type ty.

All types of higher order logic must denote non-empty sets. This means that the predicate P used in the type definition above must be true of at least one value of the representing type; i.e. it must be the case that $\vdash \exists x{:}ty.\, P\ x$. This theorem must be proved before the type definition axiom (1) can be added to the logic.

If the subset defined by P is non-empty, then adding the type definition axiom (1) shown above is a *conservative extension* of the logic. That is, for all boolean terms tm not containing the new type, $\vdash tm$ is a theorem of the extended logic exactly when it is a theorem of the original logic. In particular, $\vdash \mathsf{F}$ is a theorem of the extended logic if and only if it is a theorem of the original logic. Thus adding type definition axioms to the logic will not introduce inconsistency; adding type definition axioms is 'safe'.

In addition to type constants, new type operators can also be defined by adding axioms of the form shown above. For example, a Cartesian product type operator '$\times$' can be defined by postulating a type definition axiom of the form:

$$\vdash \exists f{:}(\alpha \times \beta){\to}(\alpha{\to}\beta{\to}bool).\ (\forall a_1\, a_2.\, f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{P}\ r = (\exists a.\, r = f\ a)) \tag{2}$$

where P is a predicate on values of type $\alpha{\to}\beta{\to}bool$, defined such that the subset of $\alpha{\to}\beta{\to}bool$ satisfying P represents the Cartesian product of the types α and β. Since (2) asserts that $(\alpha \times \beta)$ is isomorphic to this subset, adding this axiom to the logic defines '$\times$' to be the Cartesian product operation on types. See [4] for details.

2.2 Deriving Abstract Axioms for New Types

Type definition axioms of the form described above merely state that a new type is isomorphic to a particular subset of an existing type. From such type definition axioms, it is possible to derive theorems that characterize new types more abstractly. The idea is to prove a collection of theorems that state the essential properties of a new type without reference to how it is represented. These theorems can then be used for all future reasoning about the new type. The motivation for first defining a type and then *deriving* abstract 'axioms' for it is that this process guarantees consistency. Simply postulating abstract axioms for a new type may introduce inconsistency into the logic; but deriving abstract axioms from a type definition amounts to giving a formal proof of their consistency.

As an example, consider the Cartesian product type operator $\times$ defined by the type definition axiom (2) shown above. This type operator can be characterized abstractly by defining the usual projection functions

$$\mathsf{Fst}{:}(\alpha \times \beta){\to}\alpha \qquad \text{and} \qquad \mathsf{Snd}{:}(\alpha \times \beta){\to}\beta$$

such that the following theorem holds:

$$\vdash \forall f{:}\gamma{\to}\alpha.\ \forall g{:}\gamma{\to}\beta.\ \exists!\, h{:}\gamma{\to}(\alpha \times \beta).\, (\mathsf{Fst} \circ h = f) \wedge (\mathsf{Snd} \circ h = g)$$

This theorem can be derived by formal proof from the type definition axiom (2) for $\times$ and the definitions of the projection functions Fst and Snd. It is an abstract characterization of the type operator $\times$; and all the usual properties of the Cartesian product of two types follow from it, without the need to know how the type operator $\times$ is defined.

2.3 Recursive Types

In this section, a class of recursive types is described which can be defined in higher order logic using the method outlined above. The abstract 'axioms' that characterize these types can be derived from the properties of the existing types that represent them. In what follows, however, the details of these derivations will not be given; the abstract axioms for recursive types will simply be stated without proof. The process of defining these types and deriving abstract axioms for them has been automated using the HOL theorem prover for higher order logic. The details appear in [13].

2.3.1 An Example: the Type of Lists

Lists are a simple example of a recursive type. The abstract syntax of lists can be specified by the little grammar shown below:

$$\mathit{list} \quad ::= \quad \mathsf{Nil} \quad | \quad \mathsf{Cons}\ \alpha\ \mathit{list}$$

Here, Nil and Cons are the usual constructors for lists. This grammar can be seen as a 'declaration' of the recursive type of finite lists containing values of type α. It states that lists are constructed inductively from Nil using the constructor Cons.

Lists of this kind can be represented in higher order logic by defining a unary type operator $(\alpha)list$ using the type definition mechanism outlined in Section 2.1. Once a type definition axiom for $(\alpha)list$ has been introduced into the logic, an abstract axiomatization for lists can be derived based on two constants:

$$\mathsf{Nil}:(\alpha)list \qquad \text{and} \qquad \mathsf{Cons}:\alpha\rightarrow(\alpha)list\rightarrow(\alpha)list$$

The constant Nil denotes the empty list. The function Cons constructs lists in the usual way: if h is a value of type α and t is a list then $\mathsf{Cons}\ h\ t$ denotes the list with head h and tail t. Using these two constructors, the abstract axiom for $(\alpha)list$ can be written as follows:

$$\vdash \forall e\, f.\, \exists! fn.\, (fn(\mathsf{Nil}) = e) \wedge (\forall h\, t.\, fn(\mathsf{Cons}\ h\ t) = f\ (fn\ t)\ h\ t) \tag{3}$$

This theorem asserts that a function on lists $fn{:}(\alpha)list\rightarrow\beta$ can be *uniquely* defined by primitive recursion on lists—i.e. by giving a base case that defines the value of $fn(\mathsf{Nil})$, and a recursive case that defines the value of $fn(\mathsf{Cons}\ h\ t)$ in terms of $(fn\ t)$, h, and t.

All the usual properties of lists follow from the theorem shown above. For example, consider the following three theorems about lists:

$$\vdash \forall P.\, (P(\mathsf{Nil}) \wedge \forall t.\, P\ t \supset \forall h.\, P(\mathsf{Cons}\ h\ t)) \supset \forall l.\, P\ l$$

$$\vdash \forall h\, t.\, \neg(\mathsf{Nil} = \mathsf{Cons}\ h\ t)$$

$$\vdash \forall h_1\, h_2\, t_1\, t_2.\, (\mathsf{Cons}\ h_1\ t_1 = \mathsf{Cons}\ h_2\ t_2) \supset ((h_1 = h_2) \wedge (t_1 = t_2))$$

These three theorems follow immediately from the abstract axiom (3) for $(\alpha)list$. The first theorem states that properties can be proved to hold of all lists by structural induction; the second theorem states that Nil and Cons yield distinct values of type $(\alpha)list$; and the last theorem states that Cons is one-to-one. These facts mean that every value of type $(\alpha)list$ is either equal to Nil, or is constructed from Nil by finitely many applications of the constructor Cons—i.e. the set denoted by $(\alpha)list$ is the *free algebra* with constructors Nil and Cons. Thus $(\alpha)list$ denotes precisely the set of all finite-length lists of values of type α.

The abstract axiom (3) for $(\alpha)list$ can also be used to prove the existence of particular functions defined by *primitive recursion* on lists. For example, specializing the variables e and f in a suitably type-instantiated version of (3) so that:

$$e = 0 \qquad \text{and} \qquad \forall x\, y\, x.\, f\ x\ y\ z = x{+}1$$

yields the following theorem:

$$\vdash \exists! fn.\, (fn(\mathsf{Nil}) = 0) \wedge (\forall h\, t.\, fn(\mathsf{Cons}\ h\ t) = (fn\ t){+}1)$$

which asserts the (unique) existence of a *length* function defined by primitive recursion on lists. Any function definition by primitive recursion on lists can be justified in a similar way: by appropriately specializing e and f in (3).

2.3.2 General Recursive Types

The axiom (3) for lists shown above illustrates the general form of the theorems that will be used to characterize recursive types. In general, a recursive type with n constructors C_1, C_2, $\ldots$, C_n can be *informally* described by a type 'declaration' of the form:

$$rty \quad := \quad \mathsf{C}_1\ ty \cdots ty \quad | \quad \mathsf{C}_2\ ty \cdots ty \quad | \quad \cdots \quad | \quad \mathsf{C}_n\ ty \cdots ty \qquad (4)$$

where rty is the name of the recursive type being described, and each ty is either an existing logical type (not containing rty) or the name rty itself. An expression of this form is similar to a 'datatype' declaration in Standard ML [10]. It simply states the names of the constructors for a new type rty and the types of their arguments.

Any recursive type described by an informal declaration of this kind can be characterized formally in higher order logic by a single abstract axiom of the following general form:

$$
\begin{array}{l}
\forall f_1\ f_2\ \cdots\ f_n.\ \exists!\mathit{fn}{:}\mathit{rty}{\to}\alpha. \\
\quad \begin{array}{rcl}
\forall x_1 \cdots x_i.\ \mathit{fn}(\mathsf{C}_1\ x_1\ \cdots\ x_i) & = & f_1\ (\mathit{fn}\ x_1)\ \cdots\ (\mathit{fn}\ x_i)\ x_1\ \cdots\ x_i\ \wedge \\
\forall x_1 \cdots x_j.\ \mathit{fn}(\mathsf{C}_2\ x_1\ \cdots\ x_j) & = & f_2\ (\mathit{fn}\ x_1)\ \cdots\ (\mathit{fn}\ x_j)\ x_1\ \cdots\ x_j\ \wedge \\
\vdots & & \\
\forall x_1 \cdots x_k.\ \mathit{fn}(\mathsf{C}_n\ x_1\ \cdots\ x_k) & = & f_n\ (\mathit{fn}\ x_1)\ \cdots\ (\mathit{fn}\ x_k)\ x_1\ \cdots\ x_k
\end{array}
\end{array}
$$

where the right hand sides of the equations include terms $(\mathit{fn}\ x)$ only for variables x of type rty. (See, for example, the axiom for lists shown above.)

A theorem of this form states the unique existence of primitive recursive functions defined by cases on the constructors C_1, C_2, ..., C_n. From a theorem of this kind, it is possible to prove:

- a structural induction theorem for the recursive type rty,
- that the constructors C_1, C_2, ..., C_n yield distinct values of type rty,
- that each constructor C_1, C_2, ..., C_n is one-to-one, and
- the validity of any function definition by primitive recursion on rty.

Abstract axioms for recursive types (and the facts listed above, which follow from these axioms) are used in the formal proofs of the hardware verification examples given in Sections 3 and 4. Details will not, however, be given of the abstract axioms for the recursive types used; they will be simply described informally by type 'declarations' of the general form illustrated by (4). Details of the justification of structural induction and function definitions by primitive recursion for these types will also be omitted.

The definition and axiomatization of recursive types in higher order logic has been mechanized by a package written for the HOL theorem prover. Using this package, any recursive type of the kind described in this section can be constructed completely automatically from an informal type declaration of the general form illustrated by (4) above. The system reads such a declaration, finds an appropriate existing type to represent the new type, asserts a type definition axiom, and derives an abstract axiom for the new type by formal proof. The package also includes tools for deriving structural induction and for automating primitive recursive definitions. A full description of this package appears in [8, 13].

3 Reasoning about Tree-Shaped Devices

Many hardware devices can be built by wiring basic components together to make tree-shaped structures. This section contains an example which shows how recursive types of the kind described in Section 2 can be used to help specify and reason formally about such devices in higher order logic.

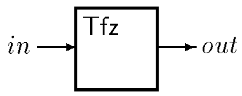
The idea is to model a collection of tree-structured devices by a recursive type whose values have the same structure as the devices themselves. Consider, for example, the set of all tree-shaped circuits built from 2-input OR-gates. The structural aspect of these circuits can be modelled in logic by an abstract type *tree* that denotes the set of all binary trees. With this approach, each value t of type *tree* represents a tree of 2-input OR-gates, and the type *tree* itself represents the set of all such circuits.

Representing circuits in this way makes it possible to specify their behaviour formally by defining a higher-order function $\mathsf{Beh}(t)$ which, for each tree t, yields a predicate defining the behaviour of the corresponding circuit. With such a parameterized specification, properties can be proved about the set of all tree-shaped circuits by showing that they follow from $\mathsf{Beh}(t)$ for all trees t.

In what follows, an example is given to illustrate these ideas. First, a formal specification is given for the intended behaviour of a simple test-for-zero device. A recursive type of trees is then used to specify a class of alternative tree-shaped circuits that implement this device. These circuits are then proved correct with respect to the specification of intended behaviour. Finally, the type of trees is used to generate provably 'optimal' circuits that satisfy the behavioural specification.

3.1 The Example Device and its Top-level Specification

A top-level view of the device discussed in this section is shown below:



This device takes an n-bit word as input on the port labelled *in*. The output on the port labelled *out* is a boolean value that indicates if the natural number represented by the n-bit word *in* is equal to zero.

To specify the behaviour of this device in logic, a data type $wordn$ is needed to model the set of all bit-vectors, or n-bit words. The following informal recursive type declaration describes such a type:

$$wordn \quad := \quad \mathsf{Wire}\ bool \quad | \quad \mathsf{Bus}\ bool\ \ wordn$$

As was discussed in Section 2.3.2, the recursive type $wordn$ described by this informal declaration can be characterized formally in logic using two primitive constructors:

$$\mathsf{Wire} : bool{\rightarrow}wordn \quad \text{and} \quad \mathsf{Bus} : bool{\rightarrow}wordn{\rightarrow}wordn$$

The function Wire takes a value b of type $bool$ and yields a one-bit word, '$\mathsf{Wire}\ b$', containing the single bit b. The function Bus takes a boolean value b and an n-bit word w and yields the $(n{+}1)$-bit word '$\mathsf{Bus}\ b\ w$' with least significant bit b. The 4-bit word 1101, for example, is denoted by

$$\mathsf{Bus\ T\ (Bus\ F\ (Bus\ T\ (Wire\ T)))}.$$

Using the recursive type $wordn$, the behaviour of the test-for-zero device can be specified in logic by a two-place predicate Tfz defined by:

$$\mathsf{Tfz}(in, out) \equiv (out = (\mathsf{Val}\ in = 0))$$

where the variable in has type $wordn$, and the variable out has type $bool$. The function $\mathsf{Val}{:}wordn{\rightarrow}num$ maps n-bit binary words to natural numbers. It is defined by primitive recursion as follows:

$$\begin{aligned} \mathsf{Val}(\mathsf{Wire}\ b) &= (b \Rightarrow 1 \mid 0) \\ \mathsf{Val}(\mathsf{Bus}\ b\ v) &= (2 \times (\mathsf{Val}\ v)) + (b \Rightarrow 1 \mid 0) \end{aligned}$$

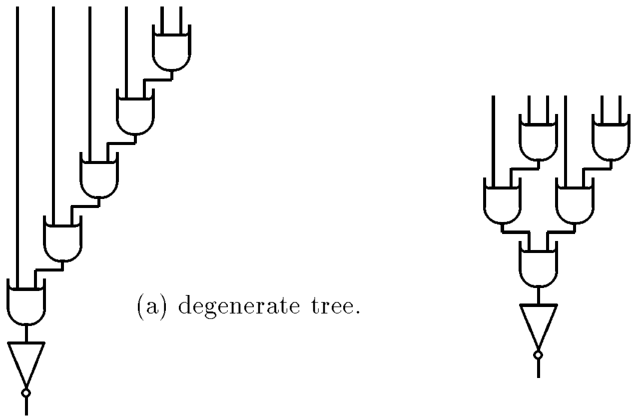
The predicate Tfz is an example of a formal specification of a class of related hardware devices. It specifies the behaviour of an n-bit test-for-zero device for all input word widths n. The actual width of the device specified by Tfz is implicitly determined by the size of the n-bit word represented by in. A similar technique for specifying n-bit wide devices is used by Hunt in [11].

3.2 Alternative Implementations of the Device

An n-bit test-for-zero device can be implemented by a tree of 2-input OR-gates connected to an inverter. Figure 1 shows two correct implementations of this kind for a 6-bit test-for-zero device. Each of these devices takes a 6-bit binary word on its input wires. The output of each device is a boolean value which is *true* if the binary number represented by its 6-bit input is zero, and *false* otherwise.

Both of the circuits shown in Figure 1 are functionally correct. The circuit on the right (circuit b), however, is an *optimal* tree-shaped implementation for a 6-bit device, in the sense that the length of the path from inputs to output is the shortest possible, and the gate delay through the device is therefore minimal. On the other hand, the structure of circuit (a) is that of a 'degenerate' binary tree: its structure is essentially that of a linear list. If the criterion by which circuits are judged is that of minimising delay, this structure is the worst implementation of a 6-bit test-for-zero device.

More generally, the best implementation of an n-bit Tfz device using a binary tree of 2-input OR-gates will be a tree of height $\lceil \log_2 n \rceil$, and the worst implementation will be a degenerate tree of height $n-1$. In Section 3.4, a recursive type *tree* of binary trees will be used to specify formally the class of *all* such implementations, and this recursive type will be used in Section 3.6



(a) degenerate tree.

(b) optimal tree.

Figure 1: Two 6-bit Test-for-zero Circuits.

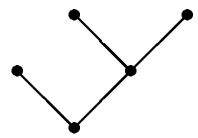
to define an 'executable' function which generates a provably optimal implementation for any input width.

3.3 A Type of Binary Trees

A recursive type $tree$ of binary trees can be described informally by:

$$tree \;\; ::= \;\; \mathsf{Leaf} \;\;|\;\; \mathsf{Node}\ tree\ tree$$

The type $tree$ has one primitive constant value, $\mathsf{Leaf}{:}tree$, and one primitive constructor $\mathsf{Node}{:}tree{\to}tree{\to}tree$. The constant Leaf denotes the trivial tree consisting of a single leaf node. The constructor Node is used to build binary trees from smaller binary trees; if t_1 and t_2 are trees, then the expression $(\mathsf{Node}\ t_1\ t_2)$ denotes the binary tree with left subtree t_1 and right subtree t_2. Using Leaf and Node, it is possible to construct a binary tree of any shape. For example, the binary tree:



is denoted by the expression: Node Leaf (Node Leaf Leaf).

Two recursive functions on trees will be used in the sections that follow. The first of these is the function $\mathsf{Height}{:}tree{\to}num$, which computes the height of a binary tree. It can be defined by primitive recursion on trees as follows:

$$\begin{aligned} \mathsf{Height\ Leaf} &= 0 \\ \mathsf{Height}\ (\mathsf{Node}\ t_1\ t_2) &= (\mathsf{Max}\ (\mathsf{Height}\ t_1)\ (\mathsf{Height}\ t_2)) + 1 \end{aligned}$$

As discussed in Section 2.3.2, the validity of a primitive recursive definition of this kind follows directly from the abstract axiom for the recursive type $tree$.

The second function on trees that will be needed below is $\mathsf{Leaves}{:}tree{\to}num$. This function computes the number of leaf nodes in a tree, and its primitive recursive definition is simple:

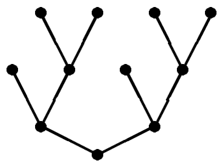
$$\begin{aligned} \mathsf{Leaves\ Leaf} &= 1 \\ \mathsf{Leaves}\ (\mathsf{Node}\ t_1\ t_2) &= (\mathsf{Leaves}\ t_1) + (\mathsf{Leaves}\ t_2) \end{aligned}$$

Again, the validity of this definition follows directly from the abstract axiom for $tree$.

3.4 Specifying the Set of all Implementations

Using the recursive type *tree* described above, it is possible to define a predicate that specifies the class of all tree-structured implementations of an n-bit test-for-zero device. The recursive type *tree* has values with the same kind of structure as the circuits to be specified. The formal specification will therefore be a predicate $\mathsf{Tfz\_imp}\ t\ (in, out)$ that is parameterized by a value t of type *tree*. For each tree t, the expression $\mathsf{Tfz\_imp}\ t\ (in, out)$ will specify the behaviour of the corresponding tree-shaped implementation. Since t ranges over the set of all binary trees, the parameterized predicate $\mathsf{Tfz\_imp}\ t\ (in, out)$ will specify the class of all such circuits.

The binary tree supplied as the parameter t to $\mathsf{Tfz\_imp}\ t\ (in, out)$ determines the shape of the tree of OR-gates in the implementation being specified. Each internal node of the tree t corresponds to an OR-gate in the implementation; and each leaf node of t corresponds to a wire of the n-bit input word. For example, if t is the binary tree:



then the circuit specified by $\mathsf{Tfz\_imp}\ t\ (in, out)$ will be the optimal 6-bit test-for-zero implementation shown in Figure 1.

3.4.1 Defining the Predicate Tfz_imp

The implementation predicate Tfz_imp is defined using an auxiliary predicate Or_tree that specifies the behaviour of the tree of OR-gates, and a predicate Inv that specifies the behaviour of an inverter. Given these two predicates, the definition of Tfz_imp is simply:

$$\mathsf{Tfz\_imp}\ t\ (in, out) \equiv \exists x.\ \mathsf{Or\_tree}\ t\ (in, x) \land \mathsf{Inv}(x, out)$$

Here, the internal line x is used to connect the tree of OR-gates defined by Or_tree to the output inverter Inv.

The definition of $\mathsf{Or\_tree}\ t\ (in, x)$ is done by primitive recursion on the tree t. In the base case, the predicate $\mathsf{Or\_tree}$ just models a direct connection between the input in (which must be a bit-vector of width one) and the output x:

$$\mathsf{Or\_tree\ Leaf}\ (in, x) \equiv (in = \mathsf{Wire}\ x)$$

In the recursive case, the predicate $\mathsf{Or\_tree}$ connects two subtrees together using a 2-input OR-gate, and then concatenates the input words of these subtrees to get the input word of the whole device:

$$\begin{aligned}&\mathsf{Or\_tree}\ (\mathsf{Node}\ t_1\ t_2)\ (in, x) \equiv \\ &\qquad \exists i_1\ i_2\ x_1\ x_2.\ (in{=}i_1 +\!\!+\ i_2) \wedge \mathsf{Or}(x_1, x_2, x) \wedge \\ &\qquad\qquad\qquad \mathsf{Or\_tree}\ t_1\ (i_1, x_1) \wedge \mathsf{Or\_tree}\ t_2\ (i_2, x_2)\end{aligned}$$

The infix operator $+\!\!+$ in this definition is the concatenation operation on n-bit words represented by the recursive type *wordn*. Its primitive recursive definition is:

$$\begin{aligned}(\mathsf{Wire}\ b) +\!\!+\ w &= \mathsf{Bus}\ b\ w \\ (\mathsf{Bus}\ b\ w_1) +\!\!+\ w_2 &= \mathsf{Bus}\ b\ (w_1 +\!\!+\ w_2)\end{aligned}$$

3.5 The Proof of Correctness

The predicate $\mathsf{Tfz\_imp}$ defines the class of all test-for-zero implementations, both for all possible shapes of the OR-gate tree and for all possible widths of the n-bit input in. Before going on to prove that this class of devices is correct with respect to the specification Tfz, it is worth checking that the predicate $\mathsf{Tfz\_imp}\ t\ (in, out)$ does not specify an inconsistent implementation, and can be satisfied for sensible values of t and in.

The following theorem asserts that $\mathsf{Tfz\_imp}$ consistently defines a circuit for all appropriate combinations of in and t:

$$\vdash \forall t\ in.\ (\exists out.\ \mathsf{Tfz\_imp}\ t\ (in, out)) \equiv (\mathsf{Leaves}\ t = \mathsf{Width}\ in)$$

This theorem states that $\mathsf{Tfz\_imp}\ t\ (in, out)$ can be satisfied by some value of out exactly when the number of leaves in the tree t matches the width of the input in. This means that $\mathsf{Tfz\_imp}$ at least defines *some* consistent and satisfiable implementation for every appropriate tree shape t. The proof of this theorem is done by structural induction on the variable t ranging over trees. As discussed in Section 2.3.2, the validity of such inductive proofs follows formally from the abstract axiom for the recursive type *tree*.

It remains to show that every test-for-zero implementation described by $\mathsf{Tfz\_imp}$ is functionally correct. The desired correctness statement is:

$$\vdash \forall t\ in\ out. \mathsf{Tfz\_imp}\ t\ (in, out) \supset \mathsf{Tfz}(in, out)$$

This theorem follows easily by structural induction on the binary tree t. It states that every n-bit implementation of a test-for-zero device constructed from an appropriate binary tree t of OR-gates satisfies the specification Tfz. Parameterizing the implementation description with the variable t of type *tree* effectively 'quantifies' over all possible shapes that the implementation can have. The correctness statement given above therefore asserts that *every* such implementation is functionally correct with respect to the abstract specification given by Tfz. The correctness statement also asserts the correctness of these designs for every possible width of the n-bit input in.

3.6 Generating Provably Optimal Implementations

In this section, a function is defined in logic to generate a provably optimal tree-shaped test-for-zero circuit (i.e. a circuit of least height) for any input word width. This is done by defining a function $\mathsf{Gen}{:}num{\rightarrow}tree$ such that $\mathsf{Gen}(n)$ denotes a binary tree of minimal height having exactly n leaf nodes. The resulting binary tree can then be used as the parameter t in the circuit specification $\mathsf{Tfz\_imp}\ t\ (in, out)$ to yield an optimal implementation. That is, given the function Gen, an optimal test-for-zero implementation for any input word in can be specified by:

$$\mathsf{Tfz\_imp}\ (\mathsf{Gen}(\mathsf{Width}\ in))\ (in, out)$$

The function Gen is defined such that it satisfies the following recursive equation for all n:

$$\vdash \mathsf{Gen}\ n = (n{<}2\ \Rightarrow\ \mathsf{Leaf} \mid \mathsf{Node}\ (\mathsf{Gen}(n\ \mathsf{div}\ 2))\ (\mathsf{Gen}(n{-}(n\ \mathsf{div}\ 2)))) \quad (5)$$

It is not, however, possible to define Gen simply by writing down the recursive equation shown above. This is because all functions in higher order logic must be total, and it is not in general the case that an arbitrary recursive equation '$f\ n = \cdots f \cdots$' involving a function variable f can in fact be satisfied by a total function. Equation (5) must therefore be *derived* in the logic by formal proof. This can be done by defining a class of functions $\mathsf{Gen}_m(n)$, each of which satisfies the recursive equation for Gen for all $n{\leq}m$, and then defining Gen by the non-recursive equation: $\forall n. \mathsf{Gen}\ n = \mathsf{Gen}_n(n)$.

The form of equation (5) is such that the value of $\mathsf{Gen}(n)$ can be computed for any particular value of n simply by rewriting with the equation and simplifying the arithmetical expressions that occur on the right hand side. This logical equation therefore acts like an executable 'functional program' for generating the optimal tree for any n, and thus for generating optimal implementations of the test-for-zero device.

It follows from the recursive equation (5) that the binary tree denoted by $\mathsf{Gen}(n)$ has n leaf nodes and is of minimal height. This is stated formally by the following two theorems:

$$\vdash \forall n.\,(n \neq 0) \supset (\mathsf{Leaves}(\mathsf{Gen}\ n) = n)$$

$$\vdash \forall t\,n.\,(\mathsf{Leaves}\ t = n) \supset \mathsf{Height}(\mathsf{Gen}\ n) \leq \mathsf{Height}\ t$$

The first of these theorems follows by mathematical induction on n; it states that Gen generates trees with the required number of leaf nodes. The second theorem can be proved by structural induction on binary trees; it states that the height of the tree denoted by $\mathsf{Gen}(n)$ is no larger than the height of any tree with n leaves. These theorems show that Gen can be used to generate optimal (and correct) implementations of the test-for-zero device.

3.7 Discussion

In the example given above, the recursive structure of the type *tree* exactly mirrors the recursive structure of the set of circuits being specified. The type of binary trees can therefore be used to model the abstract structure of these circuits independently of their behaviour. Using the recursive type *tree* in this way makes it possible to 'quantify' over the set of alternative circuits by quantifying over variables of type *tree*. This allows assertions about *all* such circuits to be formulated directly in the logic. Two assertions of this kind were presented above: (1) that every alternative implementation is functionally correct, and (2) that the circuit generated by Gen is optimal.

In the following section, this idea of modelling the structure of circuits by a recursive type is applied to reasoning about models of MOS transistor behaviour. Again, a specially-defined recursive type will be used to make it possible to formulate assertions about 'all circuits' directly as theorems of higher order logic.

4 Comparing Two Transistor Models

The value of a formal proof of correctness depends on how accurately the underlying model of circuit behaviour reflects reality. The more accurate the model, the less likely it is that design errors will escape discovery by formal proof. But a very accurate model of device behaviour may, in some cases, be unnecessarily complex. It may be possible to adopt a circuit design style for which a simpler model will do. For example, the *functional* correctness of a fully complementary CMOS circuit does not critically depend on transistor size ratios [14, pp 160–61]. A very accurate transistor model, which took into account transistor sizes, would therefore be inappropriate for this conservative circuit design style.

Assertions about the relationship between an accurate transistor model and a simpler one can be formulated naturally and concisely in higher order logic using recursive types. In this section, a specially-defined recursive type '*circ*' is introduced to provide an explicit representation in logic for the structure of the class of all CMOS circuit designs. The motivation for introducing this type is that it makes it possible for assertions about the relationship between two transistor models to be stated as *theorems* of higher order logic, rather than meta-theorems about provability in the logic. In particular, it makes it possible to prove a theorem which states that certain circuits can be verified using a simple transistor model rather than a more accurate (but also more complex) transistor model.

In the sections that follow, two simple transistor models are described. The two models are then formalized in higher order logic by means of semantic functions defined on a recursive type *circ* of MOS circuits. The relationship between these two transistor models is then formalized by a theorem which describes a condition under which the two models are effectively equivalent.

4.1 A Switch Model of Transistors

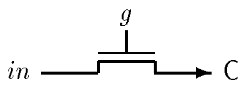
One model of transistor behaviour treats a transistor as an ideal bidirectional switch controlled by the boolean value on its gate [1, 6]. In this simple model the behaviour of a N-type transistor is specified formally as follows:

$$\mathsf{Ntran}(g, s, d) \equiv (g \supset (s = d))$$

In this specification the signals on the gate (g), source (s) and drain (d) of a transistor are modelled by values of type *bool*. If g is equal to T then the source and drain will have the same boolean value; and if g is equal to F then the source and drain can have any two boolean values. Thus Ntran acts as an ideal switch which is closed when $g{=}\mathsf{T}$ and open when $g{=}\mathsf{F}$. The specification of a P-type transistor is similar:

$$\mathsf{Ptran}(g, s, d) \equiv (\neg g \supset (s = d))$$

Although this very simple transistor model can be useful for some purposes, it fails to capture many aspects of the behaviour of real MOS transistors. One of these aspects is the fact that the switching behaviour of a MOS transistor depends not on the 'logic level' present on its gate, but on the magnitude of the gate-to-source voltage V_{gs} compared to some non-zero threshold voltage V_t. This means that MOS transistors do not act as ideal switches that pass all logic levels equally well. Consider, for example, the transmission of logic levels through an N-type transistor to a capacitative load C:



Suppose the gate g is connected directly to power. If the input in is at logic level F then any charge at C will be drained off through the transistor, and the point C will also have logic level F. But if the input is at logic level T, the voltage at point C will only reach a level that is the threshold voltage V_t less than the voltage represented by T. This voltage may be too low to drive the gate of another transistor, and therefore must be treated as distinct from the logic level T. The switch model of transistors given above does not reflect this threshold switching behaviour of real transistors; and this inadequacy makes it possible to prove in this model the 'correctness' of certain circuits which do not work in practice (an example is given below in Section 4.7).

The fundamental problem with the simple switch model is that it specifies the behaviour of transistors in terms of the two-valued type *bool*, so that each node of a circuit has either the value T or the value F. For example, in the switch model the value at point C of the transistor shown above must be either T or F. This means that when the value at in is T and the gate is high, the value at C must also be T. No other value is possible. But in a real N-type transistor, the value at C will be some 'degraded' logic level which must be treated as distinct from T. To model such degraded logic levels, a type with more than two values is needed.

4.2 A Threshold Switching Model of Transistors[2]

A transistor model that at least partly captures threshold switching behaviour can be based on an abstract data type that has exactly three distinct values: Hi, Lo, and X. Using the informal notation introduced in Section 2.3.2, an appropriate logical type '*tri*' is defined by the equation shown below.

$$tri \quad := \quad \mathsf{Hi} \quad | \quad \mathsf{Lo} \quad | \quad \mathsf{X}$$

This informal definition states that *tri* denotes a set which contains exactly three distinct values—namely Hi, Lo, and X. The abstract characterization of this three-valued type consists of the following single theorem.

$$\vdash \forall a\, b\, c.\, \exists!\mathit{fn}{:}tri{\rightarrow}\alpha.\, (\mathit{fn}\ \mathsf{Hi} = a) \wedge (\mathit{fn}\ \mathsf{Lo} = b) \wedge (\mathit{fn}\ \mathsf{X} = c) \tag{6}$$

This theorem provides a complete and abstract characterization of the defined logical type *tri*. It takes the form of a degenerate 'primitive recursion' theorem. Since *tri* is an enumerated type with no recursive constructors, the theorem simply states that any function defined by cases on the three constants Hi, Lo, and X exists and is uniquely defined. It follows immediately that the type *tri* denotes a set containing precisely the three distinct values denoted by these constants.

4.2.1 The Model

Once the type *tri* has been defined, it can be used as the basis for a transistor model which at least partly captures the threshold switching behaviour of real CMOS devices. The basic idea is to represent the strongly-driven logic levels *high* and *low* by the values Hi and Lo, and to represent all degraded logic levels, which cannot reliably drive the gates of transistors, by the value X.

The behaviour of an N-type transistor in this model is given by:

$$\mathsf{Ntran}(g, s, d) \equiv (g{=}\mathsf{Hi} \supset ((s{=}\mathsf{Lo}) = (d{=}\mathsf{Lo})))$$

It follows from this definition that when the gate g has value Hi and the source s has value Lo (i.e. the gate-to-source voltage is large) then the drain d must also have value Lo. Thus the logic level Lo is transmitted unchanged through

[2]The transistor model described in this section is based on an idea suggested by Mike Fourman at the workshop on *Theoretical Aspects of VLSI Architectures* at the University of Leeds in September 1986.

the transistor. But when both g and s have the value Hi then the value at d may be either Hi or X; the predicate Ntran is satisfied in both cases. This reflects the fact that a logic level Hi can be degraded to an 'error' value X when it is transmitted through the transistor.

The behaviour of a P-type transistor is specified in the threshold switching model as follows:

$$\mathsf{Ptran}(g, s, d) \equiv (g{=}\mathsf{Lo} \supset ((s{=}\mathsf{Hi}) = (d{=}\mathsf{Hi})))$$

Again, the value X is used to model the possibility of a logic level being degraded as it is transmitted through a transistor. In this case, when g and s are Lo the value of d can be either Lo or X, reflecting the fact that the logic level *low* is only imperfectly transmitted through a P-type transistor.

By modelling the values that can appear on the nodes of a circuit with the three-valued type *tri* instead of the two-valued type *bool*, the threshold switching model of transistors reflects the behaviour of real transistors more accurately than the simple switch model.[3] Design errors are therefore less likely to escape discovery by formal verification when the threshold switching model is used, since a circuit that can be proved correct using the switch model may be shown to be incorrect using the more accurate threshold switching model.

For certain circuits, however, the two models are effectively equivalent. For *these* circuits, a proof of correctness in the switch model amounts to a proof of correctness in the threshold switching model. The switch model is therefore an adequate basis for verification of these circuits, and the extra accuracy of the threshold switching model is not needed. In the sections that follow, a recursive type *circ* of MOS circuits is used to express formally the assertion that the two models are equivalent for a certain class of circuit designs.

4.3 A Recursive Type for the Syntax of MOS Circuits

Following the approach of Cardelli [2] and Winskel [16], a language is defined in this section whose expressions are *circuit terms* which describe how circuits are constructed from subcircuits. A circuit term is either a primitive expression that denotes a basic component (i.e. power, ground, an N-type transistor, or a P-type transistor), or a composite expression that denotes a circuit built up by the operations of *composition* (connecting two circuits together) and *hiding* (insulating wires from the environment).

[3] Of course, the threshold switching model defined in this section is still a very crude model of CMOS transistor behaviour. For a better model, see [15].

The syntax of circuit terms is represented in logic by the recursive type *circ*, informally described by:

$$\begin{array}{rcl} circ & := & \mathsf{Pwr}\ str \\ & | & \mathsf{Gnd}\ str \\ & | & \mathsf{Ntran}\ str\ str\ str \\ & | & \mathsf{Ptran}\ str\ str\ str \\ & | & \mathsf{Join}\ circ\ circ \\ & | & \mathsf{Hide}\ str\ circ \end{array}$$

where *str* is an appropriately-defined (recursive) type of ascii character strings.

This equation defines a recursive type with six constructors, corresponding to the six different syntactic constructs in the abstract syntax of the language it represents. The first four constructors represent the primitive CMOS devices power, ground, N-type transistors, and P-type transistors. These are simply functions that map wire names (modelled by strings) to values of type *circ*. The constructor $\mathsf{Pwr}{:}str{\to}circ$, for example, maps a value s of type *str* to a circuit term '$\mathsf{Pwr}\ s$' which represents a power node. Similarly, the constructor for N-type transistors, $\mathsf{Ntran}{:}str{\to}str{\to}str{\to}circ$, maps three strings to a value of type *circ* that represents an N-type transistor. For example, a circuit term of the form '$\mathsf{Ntran}\ g\ s\ d$' stands for an N-type transistor with gate labelled by the string g, source labelled by the string s, and drain labelled by the string d.

The other two constructors, Join and Hide, represent composition and hiding operations, which are used to construct circuit terms that model composite CMOS circuit designs. These two constructors are recursive functions that map values of type *circ* to values of type *circ*. The function $\mathsf{Join}{:}circ{\to}circ{\to}circ$ represents the composition operation on circuit terms. If c_1 and c_2 are two values of type *circ*, then the circuit term '$\mathsf{Join}\ c_1\ c_2$' represents the composition of the two circuits represented by c_1 and c_2. The function $\mathsf{Hide}{:}str{\to}circ{\to}circ$ represents the hiding operation on circuit terms. If c is a circuit term and s is a string, then the circuit term '$\mathsf{Hide}\ s\ c$' represents the circuit obtained by hiding the wire labelled s in the circuit represented by c.

As was outlined in Section 2.3.2, the recursive type *circ* described informally by the grammar shown above can be defined formally in higher order logic using the rule of type definition described in Section 2.1. And an abstract characterization of the form shown in Section 2.3.2 can be proved for the type *circ*, from which one can derive structural induction and primitive recursive definitions. These facts are used in the sections that follow.

4.4 The Semantics of Circuit Terms

The recursive type *circ* defined in the previous section denotes a set of values whose structure mirrors the way in which CMOS circuits are built up from their primitive components. This provides an embedded language of circuit terms in higher order logic for modelling the purely structural aspect of the class of all CMOS circuit designs. The following sections show how the *behaviour* of this class of circuit designs can also be modelled in logic by defining a formal semantics for this language. The semantics of circuit terms will be defined in two different ways. One of these corresponds to the switch model of transistor behaviour, and the other corresponds to the threshold switching model.

For both transistor models, the semantics of circuit terms will be based on the idea of an *environment.* An environment is a function $e{:}str{\rightarrow}ty$ that maps wire names (modelled by strings) to values. Such a function assigns a value '$e\ s$' to every external wire s of a device, and thus describes a possible pattern of communication with the 'environment' in which a device operates. In the switch model, values on the wires of a device are represented by booleans. An environment in this model is therefore a function $e{:}str{\rightarrow}bool$ which assigns a value '$e\ s$' of type *bool* to every wire name s. This associates a boolean logic level with every external wire of a CMOS device. In threshold switching model, the values present on the wires of a device are modelled by the three-valued type *tri* introduced above. In this model, an environment is a function of type $str{\rightarrow}tri$, which assigns a value of type *tri* to each external wire of a device.

Using this idea of an environment, a denotational semantics can be given to circuit terms by defining by a 'meaning' function M that maps circuit terms to predicates on environments. The precise definition of this function will depend on the model of transistor behaviour which is used, but the basic idea is to define a function M such that, for every circuit term c, the application '$\mathsf{M}\ c$' denotes a predicate which is satisfied by only those environments that represent allowable configurations of values on the wires of the circuit represented by c. For any environment e, the expression $\mathsf{M}\ c\ e$ will then be true exactly when e represents a configuration of externally observable values that could occur on the wires of the CMOS circuit represented by the circuit term c.

4.4.1 The Switch Model Semantics

The semantic function Sm for the switch model of CMOS transistor behaviour is defined by primitive recursion on $circ$ and has type $circ \to ((str \to bool) \to bool)$. When applied to a circuit term c, it yields a predicate $\mathsf{Sm}\ c$ on environments of type $str \to bool$. The primitive recursive definition of Sm is:

$$
\begin{array}{lcl}
\mathsf{Sm}\ (\mathsf{Pwr}\ p)\ e & = & (e\ p = \mathsf{T}) \\
\mathsf{Sm}\ (\mathsf{Gnd}\ g)\ e & = & (e\ g = \mathsf{F}) \\
\mathsf{Sm}\ (\mathsf{Ntran}\ g\ s\ d)\ e & = & e\ g \supset (e\ d = e\ s) \\
\mathsf{Sm}\ (\mathsf{Ptran}\ g\ s\ d)\ e & = & \neg(e\ g) \supset (e\ d = e\ s) \\
\mathsf{Sm}\ (\mathsf{Join}\ c_1\ c_2)\ e & = & \mathsf{Sm}\ c_1\ e \wedge \mathsf{Sm}\ c_2\ e \\
\mathsf{Sm}\ (\mathsf{Hide}\ s\ c)\ e & = & \exists b.\ \mathsf{Sm}\ c\ (\mathsf{subst}\ e\ b\ s)
\end{array}
$$

where $(\mathsf{subst}\ e\ b\ s)$ denotes the environment identical to e except that it assigns the value b to the string s. The definition of subst is:

$$\vdash \forall st.\ (\mathsf{subst}\ e\ b\ s)\ st = ((st = s) \Rightarrow b \mid e\ st)$$

The validity of this primitive recursive definition is justified formally by the characterization of the defined type $circ$ given by a theorem of the form shown in Section 2.3.2.

The first four equations in this definition of the function Sm define the semantics of primitive CMOS devices: power, ground, N-type transistors, and P-type transistors. Each equation states what must be true of an environment in which the corresponding component is operating. The equation for Ntran, for example, states what must be true in any environment in which an N-type transistor with gate g, source s, and drain d is placed. This equation imposes the constraint that any environment e which assigns the value T to g must also assign equal values to d and s. The three other equations define the semantics of power, ground, and P-type transistors in a similar way.

The last two equations shown above define the semantics of composition and hiding. The semantic equation for the constructor Join states that an environment e is a possible assignment of values to the wires in a composition of two circuits exactly when it is a possible assignment of values to the wires of *both* subcircuits. The equation for Hide uses existential quantification to isolate the hidden wire from the environment. It states that e is a possible environment for the circuit represented by '$\mathsf{Hide}\ s\ c$' exactly when there exists *some* environment which is allowed by the semantics of c and which differs from e only in the boolean value it assigns to the string s.

4.4.2 The Threshold Model Semantics

In the threshold switching model, signals on circuit nodes are modelled by values of type tri. The semantics of $circ$ for the threshold switching model will therefore be given by a function Tm from $circ$ to predicates on environments of type $str{\to}tri$. The function Tm is defined by primitive recursion as follows:

$$
\begin{array}{lcl}
\mathsf{Tm}\ (\mathsf{Pwr}\ p)\ e & = & (e\ p = \mathsf{Hi}) \\
\mathsf{Tm}\ (\mathsf{Gnd}\ g)\ e & = & (e\ g = \mathsf{Lo}) \\
\mathsf{Tm}\ (\mathsf{Ntran}\ g\ s\ d)\ e & = & (e\ g = \mathsf{Hi}) \supset ((e\ d = \mathsf{Lo}) = (e\ s = \mathsf{Lo})) \\
\mathsf{Tm}\ (\mathsf{Ptran}\ g\ s\ d)\ e & = & (e\ g = \mathsf{Lo}) \supset ((e\ d = \mathsf{Hi}) = (e\ s = \mathsf{Hi})) \\
\mathsf{Tm}\ (\mathsf{Join}\ c_1\ c_2)\ e & = & \mathsf{Tm}\ c_1\ e \wedge \mathsf{Tm}\ c_2\ e \\
\mathsf{Tm}\ (\mathsf{Hide}\ s\ c)\ e & = & \exists v.\ \mathsf{Tm}\ c\ (\mathsf{subst}\ e\ v\ s)
\end{array}
$$

This definition is similar to the recursive definition of the semantic function for the switch model semantics of circuit terms. The difference is that the function Tm defined here is defined for environments of type $str{\to}tri$, and the threshold switching model of CMOS behaviour is used in the defining equations for the primitive devices Pwr, Gnd, Ntran, and Ptran. The semantics of composition and hiding are the same as in the switch model.

4.5 Defining Satisfaction

The two semantic functions defined in the preceding sections can be used to formulate an assertion that describes a condition under which a correctness result obtained in the switch model amounts to a correctness result in the threshold switching model. This assertion will state when it is 'safe' use the less detailed switch model to verify a CMOS design, rather than the more accurate threshold switching model. The first step in formulating this assertion is to define what it means for a circuit design to 'satisfy' a specification of required behaviour in a given transistor model.

In the following definition of satisfaction, c is a circuit term representing a CMOS circuit design, M stands for a semantic function on circuit terms, and S is a specification of the circuit's required or intended behaviour:

$$\mathsf{Sat}\ M\ c\ S\ \equiv\ \forall e{:}str{\to}\alpha.\ M\ c\ e \supset S\ e$$

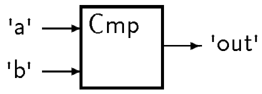
The specification S in this definition is just a predicate on environments of type $str{\to}\alpha$. The type variable α stands for the type of values on the nodes of a circuit. This type is $bool$ for the switch model semantics given by Sm

and tri for the threshold switching semantics given by Tm. The definition of Sat states that a circuit c satisfies a specification S in a transistor model M if every environment allowed by the semantics $M\ c$ is also allowed by S.

4.6 Translating Specifications

To compare the switch model and thresold switching model, a method is needed for relating Sm correctness results and Tm correctness results. Since there are assertions about correctness in the threshold model which simply cannot be expressed in the simpler switch model, the relationship must be based on a translation *from* switch model correctness statements *into* threshold model correctness statements. The basis of the comparison is therefore a translation from Sm specifications into equivalent Tm specifications.

Consider, for example, the one-bit comparator shown below:



This device compares the two bits on the input wires `'a'` and `'b'`. If they are equal, then the output `'out'` is *true*; otherwise the output is *false*. In the switch model given by Sm, the behaviour of this device can be specified by:

$$\mathsf{Cmp}\ e \equiv (e\ \texttt{'out'} = (e\ \texttt{'a'} = e\ \texttt{'b'}))$$

where `'a'`, `'b'`, and `'out'` are constants of type str, and e is an environment of type $str{\rightarrow}bool$. An equivalent Tm specification is given by:

$$\mathsf{Tcmp}\ e \equiv (\forall st.\, e\ st \neq \mathsf{X}) \supset (e\ \texttt{'out'} = ((e\ \texttt{'a'}{=}e\ \texttt{'b'}) \Rightarrow \mathsf{Hi} \mid \mathsf{Lo}))$$

where e is an environment of type $str{\rightarrow}tri$. This threshold model specification states that in a 'well-behaved' environment—that is, an environment in which no wire has the value X—the output will be Hi if the input bits are equal and Lo if they are not equal.

In this example, the threshold model specification given by Tcmp defines the same functional behaviour as the switch model specification given by Cmp. In general, any Sm specification S can be translated into an equivalent Tm specification by the function Trans, defined by:

$$\forall S\ e.\, \mathsf{Trans}\ S\ e \equiv (\forall s.\, e\ s \neq \mathsf{X}) \supset S(\mathsf{abs} \circ e)$$

where $\mathsf{abs}{:}tri{\rightarrow}bool$ is a data abstraction function defined such that $\mathsf{abs}\ \mathsf{Hi} = \mathsf{T}$ and $\mathsf{abs}\ \mathsf{Lo} = \mathsf{F}$. The definition of Trans states that the translation of an Sm specification S is true of a well-behaved Tm environment $e{:}str{\rightarrow}tri$ exactly when the specification S holds of the corresponding Sm environment $\mathsf{abs} \circ e$.

4.7 Relating the Two Models

Using the translation Trans, it is possible to formulate as theorems assertions about the relationship between the two transistor models defined by Sm and Tm. In particular, a predicate on circuit terms Wb can be defined such that one can prove:

$$\vdash \forall c.\, \mathsf{Wb}\ c \supset \forall S.\ \mathsf{Sat}\ \mathsf{Sm}\ c\ S \equiv \mathsf{Sat}\ \mathsf{Tm}\ c\ (\mathsf{Trans}\ S) \tag{7}$$

This theorem asserts that if a circuit c satisfies Wb, then it satisfies a switch model specification S exactly when it satisfies the corresponding threshold model specification $\mathsf{Trans}\ S$. The simple switch model is therefore adequate for proving the correctness of circuits which satisfy Wb. For such circuits, there is no point in using the more complex threshold switching model, since the two models exactly agree on the (Sm) specifications that these circuits satisfy.

In fact, for *any* circuit term c, a correctness result in the threshold model of CMOS behaviour implies a correctness result in the simpler switch model. By structural induction on c, one can prove:

$$\vdash \forall c\, S.\ \mathsf{Sat}\ \mathsf{Tm}\ c\ (\mathsf{Trans}\ S) \supset \mathsf{Sat}\ \mathsf{Sm}\ c\ S$$

It is necessary to impose the condition Wb on circuit terms only to prove the converse implication. This result is exactly what one would expect; for if a circuit can be proved correct using the detailed threshold model, then it must also be correct according to the simpler—but less accurate—switch model.

The converse, however, is not true. Some circuits can be proved correct with respect to a specification S in the switch model which are not correct with respect to the corresponding specification $\mathsf{Trans}\ S$ in the threshold switching model. Formally, one can prove that:

$$\vdash \neg\, \forall c\, S.\ \mathsf{Sat}\ \mathsf{Sm}\ c\ S \supset \mathsf{Sat}\ \mathsf{Tm}\ c\ (\mathsf{Trans}\ S)$$

The CMOS circuit shown Figure 2 provides a counterexample by which this negative result can be proved. This circuit is intended to be an implementation

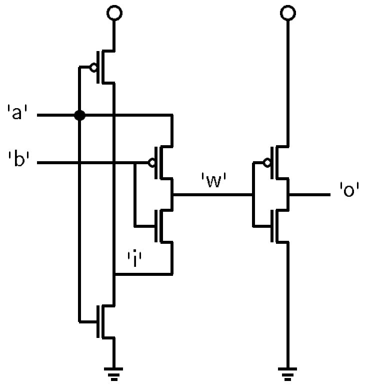


Figure 2: An Incorrect CMOS Comparator.

of the one-bit comparator specified on page 32. But it is in fact composed of an *incorrect* exclusive-or gate (the subcircuit on the left) connected to an inverter by the internal wire 'w'. The exclusive-or circuit shown in this diagram is given in [1] as an example of a CMOS design which can be proved correct using the switch model of transistors, but which is in fact incorrect due to the threshold switching behaviour of its transistors.

According to the switch model, the circuit shown in Figure 2 is a correct implementation of the one-bit comparator specified by Cmp. If Cmpr:*circ* is the circuit term that corresponds to this circuit, then one can prove:

$$\vdash \mathsf{Sat\ \ Sm\ \ Cmpr\ \ Cmp}$$

This theorem states that the implementation Cmpr is correct with respect to the switch model specification of a one-bit comparator. But according to the threshold switching model, the circuit is not correct. Formally:

$$\vdash \neg\mathsf{Sat\ \ Tm\ \ Cmpr\ \ (Trans\ Cmp)}$$

That is, the circuit Cmpr does not satisfy the corresponding threshold switching specification of a one-bit comparator.

The problem with the comparator circuit is that, for certain input values, the value on the internal wire 'w' can be the degraded logic level X because of threshold effects. If the input 'a' is Lo and the input 'b' is Hi then the value on the hidden wire 'w' can be either Hi or X. This means that the voltage on 'w' may be too low to drive the gates of the transistors in the output inverter. In the threshold model, the output 'o' is not forced to be the correct value Lo in this case, and the circuit therefore fails to satisfy the specification of required behaviour given by Trans Cmp. The problem is, of course, completely invisible to the switch model of CMOS behaviour, and the device is (incorrectly) regarded as correct in this simpler model.

The problem with the incorrect comparator circuit can be detected using the threshold model semantics because there is an environment e which satisfies the constraint Tm Cmpr e imposed by the model, but which does not satisfy the constraint Trans Cmp e imposed by the threshold switching specification. In particular, there is a threshold model environment e that satisfies the model, and makes the following assignment of values to the external wires of the device:

$$e\ \mathsf{'a'} = \mathsf{Lo}, \quad e\ \mathsf{'b'} = \mathsf{Hi}, \quad \text{and} \quad e\ \mathsf{'o'} = \mathsf{Hi}.$$

For this environment, it is not only the case that the threshold model semantics *allows* the internal wire 'w' to have the value X, but that the threshold model semantics *forces* the internal wire 'w' to have the value X.

This observation motivates the following recursive definition of the predicate Wb, which rules out circuits with internal wires that can be forced to have the value X, and therefore expresses a condition which is sufficient to make the two transistor models agree on correctness results. For the circuit terms that model primitive devices, and for circuit terms constructed using the function Join, the defining equations for the condition Wb are:

$$\begin{array}{lcl}
\mathsf{Wb}\ (\mathsf{Pwr}\ p) & = & \mathsf{T} \\
\mathsf{Wb}\ (\mathsf{Gnd}\ p) & = & \mathsf{T} \\
\mathsf{Wb}\ (\mathsf{Ntran}\ g\ so\ dr) & = & \mathsf{T} \\
\mathsf{Wb}\ (\mathsf{Ptran}\ g\ so\ dr) & = & \mathsf{T} \\
\mathsf{Wb}\ (\mathsf{Join}\ c_1\ c_2) & = & \mathsf{Wb}\ c_1 \wedge \mathsf{Wb}\ c_2
\end{array}$$

These equations simply state that the primitive devices satisfy Wb, and that a composite circuit design satisfies Wb if its subcomponents do. To rule out internal wires whose value is forced to be X for some external environment, the defining equation for Wb is:

$$\begin{aligned}\mathsf{Wb}\ (\mathsf{Hide}\ s\ c) = \mathsf{Wb}\ c\ \wedge\ \forall e.\,&(\mathsf{Tm}\ c\ e \wedge \forall st.\,\neg(st{=}s) \supset \neg(e\ st{=}\mathsf{X})) \supset\\ &\exists v.\,\neg(v{=}\mathsf{X}) \wedge \mathsf{Tm}\ c\ (\mathsf{subst}\ e\ v\ s)\end{aligned}$$

This equation states that for a circuit 'Hide s c' to satisfy the condition Wb it must be the case that the circuit c satisfies Wb, and whenever c is in an environment in which every external wire except for s does not have the value X, it is possible for the wire s not to have the value X as well. In other words, there is no well-behaved *external* environment e, which assigns only Hi or Lo to each wire name, but which forces the internal wire s to have the degenerate value X.

If the condition Wb is defined as shown above, then the following theorem about the relationship between satisfaction in the two models of transistor behaviour can be proved by structural induction on c:

$$\vdash \forall c.\,\mathsf{Wb}\ c \supset \forall S.\ \mathsf{Sat}\ \mathsf{Sm}\ c\ S \supset \mathsf{Sat}\ \mathsf{Tm}\ c\ (\mathsf{Trans}\ S)$$

This theorem states that for circuit terms c that satisfy Wb, a correctness result proved in the simple switch model implies an equivalent correctness result in the more complex threshold switching model. This expresses the fact that the simple switch model is effectively equivalent to the more detailed threshold model only for a particular class of circuit designs. For circuits satisfying Wb, the simple switch model is adequate for doing correctness proofs; the threshold switching model can not be used to detect design errors in these circuits which will not also be found by using the simpler switch model.

4.8 Discussion

The predicate Wb defined in Section 4.7 is a condition on CMOS circuit designs which is sufficient to ensure that they can be verified using the simple switch model rather than the more accurate (but also more complex) threshold model. The predicate Wb, however, was not defined in a way that makes it useful in practice for determining when the simpler switch model can be used. In the defining equations for Wb, the *semantic* function Tm is used to state the condition that hidden wires are not forced to have the value X. This means that for any particular circuit term c it is necessary to carry out a proof in the threshold model of CMOS behaviour in order to determine if the condition 'Wb c' holds. But this may be (and typically is) just as much work as simply proving a threshold model correctness theorem for the circuit represented by c. If an equivalence result of the kind stated by theorem (7) is to be useful in practice, a condition is needed that can be checked purely *syntactically*.

A syntactic condition on circuit terms that makes the two transistor models agree on correctness could be seen as a 'design rule' for CMOS circuits which ensures that they are *verifiable* using the simple switch model. One example of such a syntactic condition might be a predicate FC which is true of a circuit term c exactly when c represents a fully complementary CMOS circuit design. If such a predicate is defined formally and shown to satisfy the implication:

$$\forall c.\ \mathsf{FC}\ c \supset \forall S.\ \mathsf{Sat}\ \mathsf{Sm}\ c\ S \equiv \mathsf{Sat}\ \mathsf{Tm}\ c\ (\mathsf{Trans}\ S)$$

then a correctness proof for any circuit that satisfies the syntactic condition FC can be safely done using the simpler switch model semantics. A result of this kind would show that the switch model is adequate for fully complementary CMOS logic, and if circuits are designed using this conservative CMOS design style, the extra complexity of the threshold switching model is not needed to model them. Furthermore, if FC is a purely syntactic condition on circuit terms—i.e. a condition that describes only the *structure* of fully complementary circuit designs—then checking whether the simple switch model can be used for the correctness proof of any particular circuit design can be done without having to reason about its behaviour in the more complex threshold model.

5 Concluding Remarks

The two examples given in this paper illustrate how recursive types can be used to separate the *syntax* of circuit specifications from their *semantics*. In Section 3, a type of binary trees was used to model the structure of a class of tree shaped circuits. Each value of the type *tree* represented a corresponding tree-shaped circuit. This made it possible to formulate statements in the logic about the class of *all* such circuits. And in Section 4 a recursive type was used to model the syntax of CMOS circuits. By separating the syntax of CMOS circuits from their semantics, this recursive type made it possible to formulate assertions about the relationship between two different transistor models.

These examples show that extending the syntax of types in higher order logic to include recursive types facilitates general reasoning about not just individual devices but entire classes of devices and models of behaviour. Furthermore, *defining* recursive types using the mechanism described in Section 2.1 allows this kind to reasoning to be done formally without *ad hoc* extensions to the axioms of higher order logic, and therefore ensures that inconsistency is not introduced by adding these types.

Acknowledgements

I should like to thank the following members of the Cambridge Hardware Verification Group for their helpful comments on early drafts of this paper: Albert Camilleri, Inder Dhingra, Mike Gordon, and Jeff Joyce. Thanks are also due to Glynn Winskel for useful discussions about relating transistor models; the approach used in Section 4 is based on his paper [16], in which categorical concepts are used to reason about relationships between models of hardware. I am grateful to Gonville and Caius College for support in the form of an unofficial fellowship, during which the work described here was done.

References

[1] Camilleri, A., M. Gordon, and T. Melham, 'Hardware Verification using Higher-Order Logic', in: *From HDL Descriptions to Guaranteed Correct Circuit Designs: Proceedings of the IFIP WG 10.2 Working Conference, Grenoble, September 1986*, edited by D. Borrione (North-Holland, 1987), pp. 43–67.

[2] Cardelli, L., 'An Algebraic Approach to Hardware Description and Verification', Ph.D. dissertation, University of Edinburgh (April 1982).

[3] Church, A., 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, Vol. 5 (1940), pp. 56–68.

[4] Gordon, M., 'HOL: A Machine Oriented Formulation of Higher Order Logic', Technical Report no. 68, Computer Laboratory, University of Cambridge, revised version (July 1985).

[5] Gordon, M.J.C., 'HOL: A Proof Generating System for Higher-Order Logic', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam (Kluwer Academic Publishers, 1988), pp. 73–128.

[6] Gordon, M., 'Why higher-order logic is a good formalism for specifying and verifying hardware', in: *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, edited by G.J. Milne and P.A. Subrahmanyam (North-Holland, 1986), pp. 153–177.

[7] Gordon, M.J., A.J. Milner, and C.P. Wadsworth, 'Edinburgh LCF: A Mechanised Logic of Computation', Lecture Notes in Computer Science, Vol. 78 (Springer-Verlag, 1979).

[8] Gordon, M., et al., *The HOL System: DESCRIPTION* (DSTO and SRI International, 1989).

[9] Hanna, F.K. and N. Daeche, 'Specification and Verification using Higher-Order Logic: A Case Study', in: *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, edited by G.J. Milne and P.A. Subrahmanyam (North-Holland, 1986), pp. 179–213.

[10] Harper, R., D. MacQueen, and R. Milner, 'Standard ML', Report no. ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (March 1986).

[11] Hunt, W.A., 'The Mechanical Verification of a Microprocessor Design', in: *From HDL Descriptions to Guaranteed Correct Circuit Designs: Proceedings of the IFIP WG 10.2 Working Conference, Grenoble, September 1986*, edited by D. Borrione (North-Holland, 1987), pp. 89–129.

[12] Melham, T.F., 'Abstraction Mechanisms for Hardware Verification', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam (Kluwer Academic Publishers, 1988), pp. 267–291.

[13] Melham, T.F., 'Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic', Ph.D. dissertation, Computer Laboratory, University of Cambridge (August 1989).

[14] Weste, N.H.E. and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, VLSI Systems Series (Addison-Wesley, 1985).

[15] Winskel, G., 'Models and logic of MOS circuits', in: *Logic of Programming and Calculi of Discrete Design: International Summer School Directed by F.L. Bauer, M. Broy, E.W. Dijkstra, C.A.R. Hoare*, edited by M. Broy, NATO ASI Series, Series F, Computer and Systems Sciences, Vol. 36 (Springer-Verlag, 1987), pp. 367–413.

[16] Winskel, G., 'Relating two models of hardware', in: *Category Theory and Computer Science*, edited by D.H. Pitt, A. Poigné, and D.E. Rydeheard, Lecture Notes in Computer Science, Vol. 283 (Springer-Verlag, 1987), pp. 98–113.