A Functional Approach for Formalizing Regular Hardware Structures*

Dirk Eisenbiegler[1], Klaus Schneider[1] and Ramayya Kumar[2]

[1] Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz,
(Prof. D. Schmid), P.O. Box 6980, 76128 Karlsruhe, Germany,
e–mail:schneide@ira.uka.de
[2] Forschungszentrum Informatik, Haid–und–Neustraße 10–14, 76131 Karlsruhe, Germany,
e–mail:kumar@fzi.de

Abstract. An approach for formalizing hardware behaviour is presented which is based on a small functional programming language called primitive ML (PML). Since the basic constructs of PML are simply typed λ–terms, PML lends itself both to simulation and verification. The semantics of PML is formally embedded in higher–order logic.
The formalization scheme is based on PML–functions that allow hardware descriptions from the logical level up to the algorithmic level. Besides descriptions of real circuits, abstract forms of hardware descriptions can also be dealt with in PML. The main emphasis is thereby put on regular hardware structures which are described by means of primitive recursion. PML–descriptions can easily be converted to syntactic structures, called hardware formulae, which can then be verified by the MEPHISTO system.

1 Introduction

Embedding hardware description languages (HDLs) in a logic or some calculi is essential for verification. The semantics of such embedded HDLs, which correspond to certain formulae in the underlying formal framework, can then be used to

- verify certain properties of an implementation,
- prove equivalences of two or more implementations and
- perform correct HDL–to–HDL translations.

Existing HDLs such as VHDL and ELLA are very powerful and complex. The expressive power is an advantage for circuit design, but their semantics are not formally defined due to their complexity. Formalizing the semantics of a HDL means bridging the gap between a high level design language and the simpler elements of the logic or a particular calculus.

Functional HDLs such as ELLA are rather close to logic. Those elements which consist exclusively of λ–terms can be converted to higher–order logic almost unchanged. In contrast to functional languages, procedural languages are more difficult to be formalized due to their operational semantics. For every basic instruction, it

* This work has been partly financed by a german national grant, project Automated System Design, SFB No.358.

must be described, how the execution of the instruction changes the global state. The effect of compound instructions must be derived from the effect of the basic instructions and the control structures.

There are several research projects about formalizing existing HDLs in higher–order logic [BGGH92, BGHT91, CaGM86] or other calculi [Hunt86, BoPS92]. In contrast to these projects, the starting–point of the approach presented in this paper is not a given HDL. Instead, a simple functional language called primitive ML (PML) is built on top of the logic such that its semantics is given right away. PML will be the basis for hardware descriptions. It can be regarded as a common sublanguage of HOL and ML (figure 1) — its syntax is similar to ML and every construct has a

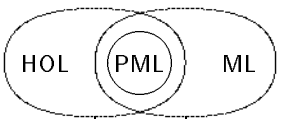


Fig. 1. Relation between HOL, ML and PML.

corresponding representation in HOL. The embedding of PML has been done in a *shallow* manner [BGGH92]. However, both PML–terms and the corresponding HOL–terms are simply typed λ–terms and there are only very small syntactical differences which could simply be overcome by pretty–printing.

Relational descriptions are frequently used for formalizing circuits, i.e. input and output signals need not be distinguished and signals can affect each other in an arbitrary manner. However, relational circuit descriptions can be ambiguous or even contradictory. In contrast, circuits are described in PML in a functional manner, i.e. circuits are represented by functions mapping input signals onto output signals.

PML cannot be viewed as a hardware description language such as VHDL, but as a general purpose programming language we use for describing hardware. In contrast to VHDL, PML has no hardware specific syntactic elements such as signals, interfaces and timing declarations. PML programmes can merely describe primitive recursive and μ–recursive functions and these functions can be regarded as a representation of the corresponding hardware.

In our hardware formalization scheme, we will describe two kinds of PML–functions: ones that represent single real circuits and others that describe sets of real circuits. Functions representing exactly one real circuit are called *concrete circuits* and functions describing a set of real circuits are called *abstract circuits*. Abstract circuits do not represent real circuits, but correspond to a scheme for describing regular hardware structures. Concrete circuits can be derived from abstract circuits by type instantiation and variable substitution.

In this paper, only the formalization of circuits is described. PML can be viewed as a more abstract layer for MEPHISTO [KuSK93, ScKK93c]. The verification of descriptions using PML is achieved by converting them into formulae which can be handled by MEPHISTO.

After having given a description of PML in section 2, concrete and abstract circuits are formalized in section 3 and 4, respectively. Finally, we briefly discuss the use of PML in simulation and verification in section 5.

2 Syntax and Semantics of PML

PML is a sublanguage of ML (figure 1). The syntax is similar, but there are less syntactical constructs — for example there are no exceptions and no side effects. Furthermore, data types in PML correspond to HOL–style data types and therefore they are weaker than ML data types [Gunt92]. Although PML has only a small number of basic elements, arbitrary μ–recursive functions can be expressed by PML functions, i.e. PML is Turing–complete.

2.1 Data Types and Primitive Recursion

In PML only HOL–style data types are allowed. For a detailed explanation of data types in HOL and the mechanism for defining them see [Melh88]. The syntax of a PML data type declaration is as follows:

```
primitive_datatype  string  ;
```

The parameter *string* defines the data type by giving the name of the type, the names of the constructors and the types of their arguments. The syntax is the same as in the HOL function define_type, e.g.

```
primitive_datatype  " bool = T | F " ;
primitive_datatype  " num  = Zero | Suc of num " ;
```

The semantics of HOL–style data types is described by a theorem which states that the primitive recursion over this type is unambiguous. In HOL, this theorem is derived whenever a new data type is introduced by define_type. A data type declaration in PML also introduces a basic function named PRIMREC_*type*, which is generated automatically. PRIMREC_*type* is derived from the semantics of the data type and it can be used for expressing primitive recursion over that data type. For example, the data types *num* and *bool* lead to the functions PRIMREC_bool and PRIMREC_num, respectively. They have the following semantics:

$$\begin{aligned} &\mathsf{PRIMREC\_bool}\ \mathsf{T}\ a\ b\ =\ a \\ &\mathsf{PRIMREC\_bool}\ \mathsf{F}\ a\ b\ =\ b \end{aligned}$$

$$\begin{aligned} &\mathsf{PRIMREC\_num}\ \mathsf{Zero}\ a\ f\ \ \ \ =\ a \\ &\mathsf{PRIMREC\_num}\ (\mathsf{Suc}\ n)\ a\ f\ =\ f\ n\ (\mathsf{PRIMREC\_num}\ n\ a\ f) \end{aligned}$$

Arbitrary primitive recursive functions can be expressed by constant definitions based on PRIMREC–functions.

2.2 Derived Functions

As in ML, functions and constants can be added by the language constructs fun and val, respectively. However, function and constant definitions in PML both correspond to constant definitions in HOL. Therefore, function and constant definitions of PML are less powerful than those in ML. The restrictions are:

- There must be only one equation within a fun or val definition, e.g.
  fun is_zero 0 = T | is_zero(Suc n) = F; is not a valid PML definition.

- The parameters on the left hand side of the equation may only be variables and paired variables, e.g. val (Suc n) = y; is not allowed.
- The expressions on the right hand side are built up by function applications ($f\ a$) and λ–abstractions (fn x => a). The only basic functions are PRIMREC–functions and WHILE (WHILE will be introduced later in section 2.5).
- The function being defined must not appear on the right hand side of the equation, e.g. fun odd n = PRIMREC_num n F (fn a => fn b => not(odd a)); is not a valid PML definition. Recursion can always be expressed by equivalent definitions which use PRIMREC–functions and WHILE.
- There is no exception handling.
- (case ... of ...) has not yet been implemented.

2.3 Predefined Data Types

Some data types are already defined in order to support pretty–printing for them. For example, it is possible to write 2 instead of Suc(Suc Zero). However, there is no pretty–printing for user defined types. The predefined data types are:[1]

```
primitive_datatype  " bool = T | F " ;
primitive_datatype  " prod = Comma of 'a # 'b " ;
primitive_datatype  " list = Nil | Cons of 'a # list " ;
primitive_datatype  " num  = Zero | Suc of num " ;
```

The following syntactic sugar refers to the predefined types. They can all be put down to expressions based on data type constructors and PRIMREC–functions.

- (a, b) may be used instead of (Comma a b)
- (fn (x, y) => $p[x, y]$) may be used instead of
  (fn z => PRIMREC_prod z (fn x => fn y => $p[x, y]$))
- (let val x = p in $q[x]$ end) may be used instead of ((fn x => $q[x]$) p).
- numerals 0, 1, 2,... may be used instead of Zero, (Suc Zero), (Suc(Suc Zero)),...
- [], [a], [a, b],... may be used instead of Nil, (Cons a Nil), (Cons a (Cons b Nil)),...

2.4 Example

We illustrate the use of the language constructs by a traffic light controller. First, a new data type named *state* is defined, which represents the states of the traffic light.

```
primitive_datatype  " state = Green | Yellow | Red " ;
```

This type declaration automatically introduces the function PRIMREC_state with the following semantics:

PRIMREC_state Green $a\ b\ c$ = a
PRIMREC_state Yellow $a\ b\ c$ = b
PRIMREC_state Red $a\ b\ c$ = c

[1] In PML, type variables are expressed by 'a, 'b, 'c, etc. whereas the corresponding type variables in HOL are expressed by α, β, γ, etc. It should be noted that the polymorphism in PML corresponds to that of HOL only.

A constant named init containing the initial state Red can be defined as:

```
val init = Red;
```

The function next takes a state as parameter and calculates the successor state. In this simple traffic light controller, the state changes from red directly to green, but changes from green to red via yellow.

```
fun next x = PRIMREC_state x Yellow Red Green;
```

Figure 2 shows the entire programme and the corresponding HOL–formula describing its semantics.

PML–Program

```
primitive_datatype
 "state = Green | Yellow | Red";




val init = Red;
fun next x =
 PRIMREC_state x Yellow Red Green;
```

Semantics

$$
\begin{array}{l}
(\forall a\, b\, c.\, \exists_1 g. \\
\quad (g\,\mathsf{Green} = a) \;\wedge\; (g\,\mathsf{Yellow} = b) \wedge \\
\quad (g\,\mathsf{Red} = c)\,) \wedge \\
(\forall a\, b\, c. \\
\quad (\mathsf{PRIMREC\_state\ Green}\ a\, b\, c = a) \wedge \\
\quad (\mathsf{PRIMREC\_state\ Yellow}\ a\, b\, c = b) \wedge \\
\quad (\mathsf{PRIMREC\_state\ Red}\ a\, b\, c = c)\,) \wedge \\
(\mathsf{init} = \mathsf{Red}) \wedge \\
(\forall x.\ \mathsf{next}\ x = \\
\quad \mathsf{PRIMREC\_state}\ x\ \mathsf{Yellow\ Red\ Green}\,)
\end{array}
$$

Fig. 2. Traffic light programme.

2.5 μ–Recursion

According to Church's Thesis, there are several equivalent schemes for describing computable functions. μ–recursive functions are one means for describing computable functions. With the elements described until now, only primitive recursive functions can be described, while μ–recursive functions cannot. Primitive recursive functions are sufficient for describing hardware *implementations*, but they are too weak for formalizing algorithmic *specifications*. Previous work such as the approaches followed by the Boyer–Moore community, are limited to primitive recursive specifications that cannot express all kinds of algorithms.

Unlike primitive recursive functions, μ–recursive functions need not be total. In ML it is possible that the evaluation of a function application does not terminate. The equations of an ML function definition can be considered as a constant specification in HOL where the specified function need not be described unambiguously, i.e. nothing can be said about the value of a function application where the evaluation does not terminate. In contrast to ML, the result of a PML function always has an explicitly defined value, even if the function application does not terminate. In this case, the value of the function is explicitly defined to be the constant Undefined, otherwise the result is (Defined y) for a certain y. The data type *partial* is used for describing values of μ–recursive functions:

```
primitive_datatype  " partial = Defined of 'a | Undefined " ;
```

The corresponding function PRIMREC_partial has the following semantics:

$$\begin{aligned}&\mathsf{PRIMREC\_partial}\ (\mathsf{Defined}\ x)\ f\ a\ =\ f\ x\\&\mathsf{PRIMREC\_partial}\ \mathsf{Undefined}\ f\ a\ =\ a\end{aligned}$$

The function WHILE is the basis for μ–recursion in PML and can be used to create loops. Given functions f and g and a parameter x, it iterates f until a value x is reached with $g\ x = \mathsf{F}$.

$$\begin{aligned}&\vdash\ \mathsf{iota}\ f\ =\ \mathsf{PRIMREC\_bool}\ (\exists_1\ f)\ (\mathsf{Defined}(\varepsilon\ f))\ \mathsf{Undefined}\\&\vdash\ \mathsf{terminates}(f,n)\ =\ (f\ n)\ \wedge\ (\forall m.\ m < n\ \Rightarrow\ \neg(f\ m))\\&\vdash\ \mathsf{mu}\ f\ =\ \mathsf{iota}(\lambda m.\ \mathsf{terminates}(f,m))\\&\vdash\ \mathsf{power}\ f\ n\ x\ =\\&\qquad\mathsf{PRIMREC\_num}\ n\ (\mathsf{Defined}\ x)\ (\lambda a\ b.\ \mathsf{PRIMREC\_partial}\ b\ f\ \mathsf{Undefined})\\&\vdash\ \mathsf{WHILE}\ g\ f\ x\ =\\&\qquad\mathsf{PRIMREC\_partial}\\&\qquad\quad(\mathsf{mu}(\lambda n.\ \mathsf{PRIMREC\_partial}\ (\mathsf{power}\ f\ n\ x)\\&\qquad\qquad\qquad(\lambda y.\ \mathsf{PRIMREC\_bool}\ (g\ y)\ \mathsf{F}\ \mathsf{T})\ \mathsf{F}))\\&\qquad\quad(\lambda n.\ \mathsf{power}\ f\ n\ x)\\&\qquad\quad\mathsf{Undefined}\end{aligned}$$

The semantics of WHILE is described using four auxiliary constants: iota, terminates, mu and power. The function iota resembles the Hilbert operator. But in contrast to the Hilbert operator its value is (Defined y) in case the predicate specifies a *unique* value and Undefined if it does not. The predicate terminates(f,n) states that n is the smallest number such that $(f\ n)$ becomes true. mu is a formalization of the μ–operator where mu f = Undefined corresponds to $\mu(f)\uparrow$ [2] and mu f = Defined k corresponds to $\mu(f) = k$. The function power computes the multiple application of a function, i.e. (power f n x) computes $f^n(x)$. The expression (WHILE g f x) calculates $f^{\mu(\lambda n.\, g(f^n(x))=\mathsf{F})}(x)$. [3]

2.6 Example

The traffic light example of section 2.4 is extended by a μ–recursive function red_time. red_time is calculating the next time when the traffic light becomes red. The traffic light is described by a function $f_{num\rightarrow state}$, that is assigning a state to every time. For a given function $f_{num\rightarrow state}$ and a time t_{num}, the function red_time calculates the smallest $n \geq t$ with $f\ n = \mathsf{Red}$.

Figure 3 shows an implementation in PML in comparison with an implementation in ML. The implementations in PML and ML are not really equivalent, since their types differ. In PML the result of red_time has the type (*num partial*) whereas in ML it is *num*.

[2] $\mu(f)$ is the smallest element of $\{f(x) = \mathsf{T} | x \in N\}$ and $\mu(f)\uparrow$ denotes that $\{f(x) = \mathsf{T} | x \in N\}$ is empty.

[3] Extending primitive recursive functions by WHILE is equivalent to the extension by the μ–operator, i.e. both extensions lead to computable functions. However, WHILE–expressions can be evaluated more efficiently by an interpreter than expressions using the μ–operator.

PML–Implementation

```
fun is_not_red x =
  PRIMREC_state x T T F;

fun red_time f t =
  WHILE
    (fn n => is_not_red(f n))
    (fn n => Defined(Suc n))
    t;
```

ML–Implementation

```
fun red_time f t =
  if ((f t) = Red) then
    t
  else
    (red_time f (Suc t));
```

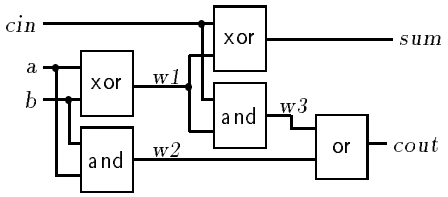
Fig. 3. Implementations of red_time.

3 Concrete Circuits

The functions described in this section are called concrete circuits since each of them corresponds to exactly one real circuit. A concrete circuit is represented by a function that assigns an input signal onto an output signal. Hardware structures are build up by function definitions. Expressing a structure by a function definition is possible, if and only if the structure does not have cycles. Since structures of sequential circuits essentially have cycles, sequential circuits are represented by a triple consisting of a combinational transient circuit, a combinational output circuit and an initial state. Instead of combining sequential circuits directly, their transient circuits, output circuits and initial states are combined.

3.1 Combinational Circuits

Individual signals of combinational circuits have type *bool*. Other than individual signals can be obtained by pairing individual signals. Combinational circuits are represented by functions assigning an input signal to an output signal. Figure 4 shows a 1–bit fulladder implemented by the circuits and, or and xor.




```
fun fulladder (cin,(a,b)) =
  let val w1 = xor(a,b) in
  let val w2 = and(b,a) in
  let val sum = xor(cin,w1) in
  let val w3 = and(cin,w1) in
  let val cout = or(w3,w2) in
    (sum,cout)
  end end end end end;
```

Fig. 4. Structure of a 1–bit fulladder and the PML representation.

3.2 Sequential Circuits

The signals of sequential circuits are time dependent. Their type is $num \rightarrow \gamma$ where the type *num* represents the discrete time and γ is the type of a time independent

signal. Sequential circuits map time dependent input signals onto time dependent output signals, thus they have the following type: $(num \rightarrow \alpha) \rightarrow (num \rightarrow \beta)$.

The description style used for combinational circuits does not allow cycles which are necessary for sequential circuits. In order to use this scheme also for sequential circuits, we define a sequential circuit by a triple (f, g, q) consisting of a combinational transient circuit f, a combinational output circuit g and an initial state q. Thus, sequential structures can be expressed by interconnecting combinational circuits. Since structures of Mealy machines might lead to zero–delay–cycles, in this paper only Moore circuits will be considered (see figure 5).

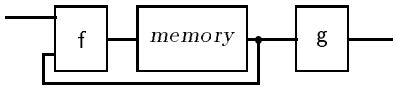


Fig. 5. Scheme of a Moore circuit.

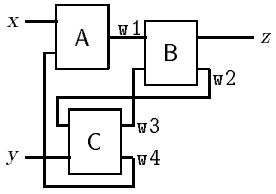
A function named makeseq is introduced, which computes a sequential circuit for a given triple (f, g, q). makeseq can be defined by the equations below. The function definition given by these equations does not have the form of a PML function definition. It can rather be regarded as a ML–style function definition or a constant specification in HOL. These equations describe the desired properties of the intended PML function in a clearer manner.

$$\begin{aligned} &\mathsf{makeseq}\ (f, g, q)\ a\ \mathsf{0} &&= f\ q \\ &\mathsf{makeseq}\ (f, g, q)\ a\ (\mathsf{Suc}\ t) &&= \mathsf{makeseq}\ \ (f, g, g(a\,t, q))\ \ (\lambda t.\, a(\mathsf{Suc}\ t))\ \ t \end{aligned}$$

The corresponding implementation in PML:

```
fun makeseq (f,g,q) a t =
   f (PRIMREC_num t q (fn n => fn r => g(a n,r)));
```

It shall be demonstrated, how structures of sequential circuits can be described in PML by function definitions. Figure 6 shows an example for a structure consisting




```
fun fD (qa,qb,qc) =
  let val (z,w2) = (fB qb) in z end;

fun gD ((x,y),(qa,qb,qc)) =
  let val (w1,(z,w2),(w3,w4)) =
        (fA qa,fB qb,fC qc)
  in
    (gA(x,w4),gB(w1,w3),gC(w2,y))
  end;

val qD = (qA,qB,qC);
```

Fig. 6. Structure of a sequential circuit and the PML representation.

of three sequential circuits A, B and C. The circuits A, B and C are represented by (fA, gA, qA), (fB, gB, qB) and (fC, gC, qC). The entire circuit is called D and its triple (fD, gD, qD) can directly be extracted from the structure.

4 Abstract Circuits

In the previous section, functions were used for describing single real circuits. In contrast to concrete circuits, abstract circuits represent sets of (concrete) circuits and are therefore more powerful than concrete circuits. Similar to concrete circuits, abstract circuits can also be represented by PML–functions. Abstract circuits can be polymorphic and allow parameters which have types that are not restricted to pairs (e.g. lists and trees may be used for instance). Concrete circuits can be obtained from abstract circuits by type instantiation and variable substitution.
The function mux is an example for an abstract circuit:

```
fun mux((s:bool),(a:'a),(b:'a)) = PRIMREC_bool s b a;
```

Concrete circuits can be derived from mux by instantiating the type variable α (α is expressed by `'a` within the PML syntax). Figure 7 shows two instances of mux.

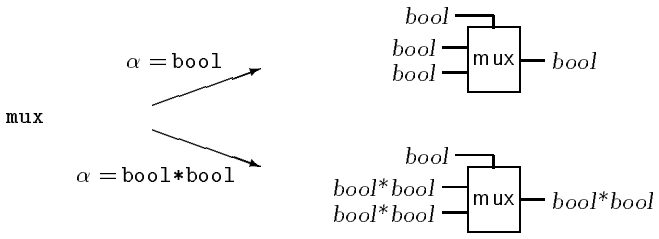


Fig. 7. Instances of a polymorphic 2:1–multiplexer.

Abstract circuits are used for formalizing regular hardware structures which can be expressed by means of primitive recursion. In general, regular circuit structures lead to regular signal structures. If for example, a structure is described, that consists of n combinational circuits connected in parallel, then it would be appropriate to use the type list for grouping together the input and the output signals.
The structure of the input signal determines the structure of the circuit and the structure of the output signal. Grouping signals together by recursive types such as *list* is flexible, since the structure of the signals and especially the number of the individual signals depends on the value.

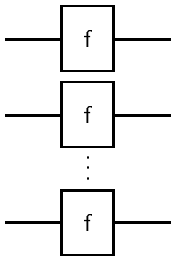


Fig. 8.: Regular circuit.

Other types than *list* can be used for grouping signals. In the next example, signals are grouped together by a list and a binary tree.

```
primitive_datatype "list = Nil | Cons of 'a # list";
primitive_datatype "btree = Bleaf of 'a | Bnode of btree # btree";
```

The semantics of the corresponding PRIMREC–functions is:

$$\begin{aligned}
&\mathsf{PRIMREC_list\ Nil}\ a\ f &&= a\\
&\mathsf{PRIMREC_list\ (Cons}\ x\ y)\ a\ f &&= f\ x\ y\ (\mathsf{PRIMREC_list}\ y\ a\ f)\\
&\mathsf{PRIMREC_btree\ (Bleaf}\ x)\ f\ g &&= f\ x\\
&\mathsf{PRIMREC_btree\ (Bnode}\ x\ y)\ f\ g &&=\\
&\quad g\ x\ y\ (\mathsf{PRIMREC_btree}\ x\ f\ g)\ (\mathsf{PRIMREC_btree}\ y\ f\ g)
\end{aligned}$$

The input of the 2^n:1–multiplexer consists of a group of data inputs and an address signal for selecting one of the data inputs. The data input signals have an arbitrary type α and they are grouped together as (α *btree*). The address signals are represented by a list of booleans (see figure 8).

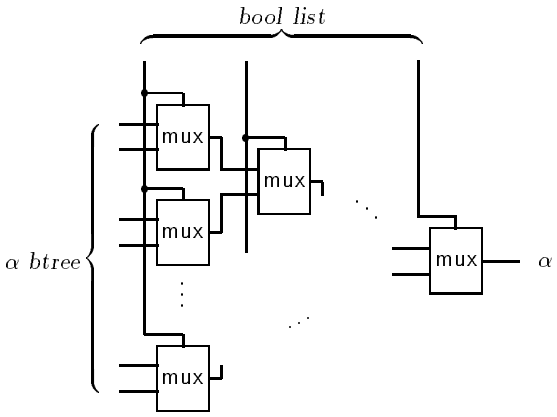


Fig. 9. 2^n:1–multiplexer.

The structure of the 2^n:1–multiplexer depends on the structure of the input signals (i.e. it depends on the length of the boolean list) and it also depends on the shape of the binary tree. The PML function representing the 2^n:1–multiplexer function is total and so the 2^n:1–multiplexer has to be designed for arbitrary lists and arbitrary binary trees, even though after instantiation the binary tree has a constant depth which is equal to the length of the list.

Figure 10 illustrates, how the the structure of the 2^n:1–multiplexer bmux is defined. For a given structure of the input signals, a circuit structure describes how the 2^n:1–multiplexer can recursively be put down to other 2^n:1–multiplexers having 'smaller' input structures in the sense of a canonical term–ordering.

The following equations give a formal definition of the description in figure 10. The equations correspond to the circuit structures of the figure in a one–to–one manner.

$$\begin{array}{lcl} \mathsf{bmux}(x, \mathsf{Bleaf}\ a) & = & a \\ \mathsf{bmux}(\mathsf{Nil}, \mathsf{Bnode}\ b\ c) & = & \mathsf{bmux}(\mathsf{Nil}, b) \\ \mathsf{bmux}(\mathsf{Cons}\ h\ t, \mathsf{Bnode}\ b\ c) & = & \mathsf{mux}(h, \mathsf{bmux}(t, b), \mathsf{bmux}(t, c)) \end{array}$$

Obviously bmux is a primitive recursive function, but these equations cannot directly be used for the PML implementation. To implement bmux in PML, the definition has to be transformed: the interlocking primitive recursions over *list* and *btree* have to be broken up:

$$\begin{array}{lcl} \mathsf{bmux}(x, \mathsf{Bleaf}\ a) & = & a \\ \mathsf{bmux}(\mathsf{Nil}, \mathsf{Bnode}\ b\ c) & = & \mathsf{bmux}(\mathsf{Nil}, b) \\ \mathsf{bmux}(\mathsf{Cons}\ h\ t, \mathsf{Bnode}\ b\ c) & = & \mathsf{mux}(h, \mathsf{bmux}(t, b), \mathsf{bmux}(t, c)) \\ & \Downarrow & \end{array}$$

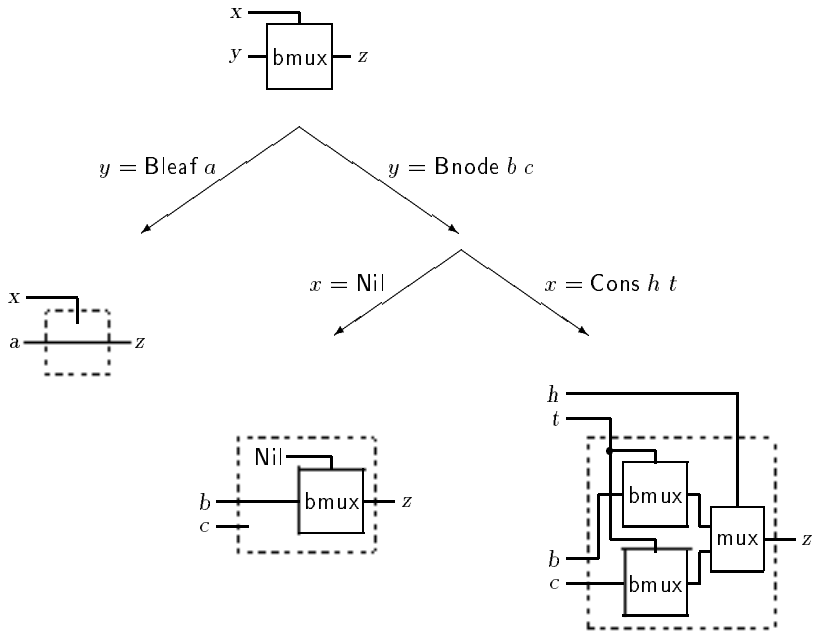


Fig. 10. Recursive description of the structure of the 2^n:1–multiplexer.

$$
\begin{array}{lcl}
\mathsf{bmux}(\mathsf{Nil}, \mathsf{Bleaf}\ a) & = & a \\
\mathsf{bmux}(\mathsf{Nil}, \mathsf{Bnode}\ b\ c) & = & \mathsf{bmux}(\mathsf{Nil}, b) \\
\mathsf{bmux}(\mathsf{Cons}\ h\ t, \mathsf{Bleaf}\ a) & = & a \\
\mathsf{bmux}(\mathsf{Cons}\ h\ t, \mathsf{Bnode}\ b\ c) & = & \mathsf{mux}(h, \mathsf{bmux}(t, b), \mathsf{bmux}(t, c)) \\
 & \Downarrow & \\
\mathsf{bmux}(\mathsf{Nil}, y) & = & \mathsf{PRIMREC\_btree}\ y\ (\lambda\, a.\, a)\ (\lambda\, a\, b\, v\, w.\, v) \\
\mathsf{bmux}(\mathsf{Cons}\ h\ t, y) & = & \mathsf{PRIMREC\_btree}\ y\ (\lambda\, a.\, a) \\
 & & (\lambda\, a\, b\, v\, w.\, \mathsf{mux}(h, \mathsf{bmux}(t, v), \mathsf{bmux}(t, w)) \\
 & \Downarrow & \\
\mathsf{bmux}(x, y) & = & \mathsf{PRIMREC\_list}\ x \\
 & & (\lambda\, s.\, \mathsf{PRIMREC\_btree}\ s\ (\lambda\, a.\, a)\ (\lambda\, a\, b\, v\, w.\, v)) \\
 & & (\lambda\, h\, t\, r\, s. \\
 & & \quad \mathsf{PRIMREC\_btree}\ s\ (\lambda\, a.\, a)\ (\lambda\, b\, c\, v\, w.\, \mathsf{mux}(h, r\, b, r\, c))\ ) \\
 & & y
\end{array}
$$

The corresponding implementation in PML is:

```
fun bmux (x,(y:'a btree)) =
  PRIMREC_list x
    (fn s =>
      PRIMREC_btree s (fn a => a)
        (fn a => fn b => fn v => fn w => v) )
    (fn h => fn t => fn r => fn s =>
      PRIMREC_btree s (fn a => a)
```

```
        (fn b => fn c => fn v => fn w => mux(h,r b,r c)) )
    y;
```

Up to now, merely regular structures of combinational circuits were considered. Regular structures of sequential circuits can also be described since sequential circuits can be put down to combinational circuits. Figure 11 shows a regular structure

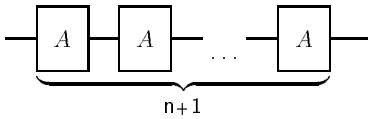


Fig. 11. A series of Moore circuits.

based on a Moore circuit called A. The function series has been implemented in PML. It takes A and n as parameters and calculates the entire circuit as shown in figure 11. Both the parameter A and the result of the function application are Moore circuits that are represented by the triple as described in the previous section.

5 Simulation and Verification

5.1 An Interpreter for PML

PML programmes can simply be executed by a ML interpreter, however, the ML environment has to be extended by some functions and data types. The type declaration construct primitive_datatype has to be implemented as a ML–function, the predefined data types of PML have to be declared and the function WHILE has to be implemented.

As all PML programmes are also ML programmes and the extended ML interpreter still accepts ML programmes, it is not tested whether the input is a PML programme or more general, a ML programme.

5.2 Simulation Tools

Some general tools for simulating circuits have been implemented in the extended ML interpreter. These tools are not PML functions, but they take PML functions which describe circuits as arguments. Moreover, they display values of output signals during the simulation of combinational and sequential circuits.

For this reason, output functions called *type*_to_string have been implemented for all predefined data types. These output functions convert a value of a certain PML type to a string. When a new data type is added, a corresponding output function should also be implemented. Output functions are used as parameters of the following simulation tools.

A function called function_table has been implemented for combinational circuits for performing the simulation and displaying the results as a table.

Sequential circuits that are represented by triples (f, g, q) could be simulated using the function makeseq. However, if the output is considered over a period, the use of make_seq would be very inefficient because for every single output, the

calculation would start from the beginning. The simulation function for sequential circuits that has been implemented does not have this disadvantage. The circuit is simulated only once until the last point of time of the considered period is reached. The parameters for a sequential simulation are: the circuit represented by (f, g, q), a time dependent input signal, a condition for terminating the simulation and an output function for converting the circuits output to a string.

5.3 Verification

A function called extend_theory_by_pml is implemented for converting a PML programme to HOL. Some tools are provided for reasoning about PML functions. They are concerned with: extending constant abbreviations, evaluation and induction.

For concrete circuits and some classes of abstract circuits, a more direct approach for verification is used. The PML–terms can be converted into certain formulae, called hardware–formulae [ScKK93c], which can be automatically verified within the MEPHISTO system [KuSK93] (see figure 12). Thus PML descriptions can also be used as a front–end specification language within this verification framework.

```
fun fulladder (cin,(a,b)) =
  let val w1 = xor(a,b) in
  let val w2 = and(b,a) in
  let val sum = xor(cin,w1) in
  let val w3 = and(cin,w1) in
  let val cout = or(w3,w2) in
    (sum,cout)
  end end end end end;
```

$\rightarrow$

$$\begin{aligned}
&\forall cin\, a\, b\, sum\, cout.\\
&\quad \mathsf{falladder}(cin, a, b, sum, cout) \Leftrightarrow\\
&\qquad \exists w1\, w2\, w3.\\
&\qquad\quad \mathsf{xor}(a, b, w1) \wedge\\
&\qquad\quad \mathsf{and}(b, a, w2) \wedge\\
&\qquad\quad \mathsf{xor}(cin, w1, sum) \wedge\\
&\qquad\quad \mathsf{and}(cin, w1, w3) \wedge\\
&\qquad\quad \mathsf{or}(w3, w2, cout)
\end{aligned}$$

Fig. 12. Converting PML circuit descriptions to hardware–formulae.

6 Conclusion and Future Work

We have presented a general purpose programming language that is formally embedded in higher–order logic and we have also demonstrated, how this language can be used for formalizing both combinational and sequential circuits. The main emphasis has been put on demonstrating how regularity can be expressed by means of primitive recursion.

PML is a very simple language and writing PML programmes can be rather tough since all function definitions have to be broken up to the primitive recursion and μ–recursion constructs. It is intended to improve the applicability of PML by adding a more comfortable ML–style mechanism for expressing recursive functions.

In future research it shall be analyzed, how PML descriptions of circuits can be used for hardware design. PML descriptions shall be used in several fields: verification, simulation, symbolic simulation, synthesis and optimization (HDL–to–HDL transformations).

References

[BGGH92] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experiences with Embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R. Boute, editors, *Conference on Theorem Provers in Circuit Design*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.

[BGHT91] R. Boulton, M. Gordon, J. Herbert, and J. van Tassel. The HOL verification of ELLA designs. In *International Workshop on Formal Methods in VLSI Design*, January 1991.

[BoPS92] D. Borrione, L. Pierre, and A. Salem. Formal verification of VHDL descriptions in Boyer-Moore: First results. In J. Mermet, editor, *VDHL for Simulation, Synthesis and Formal Proofs of Hardware*, pages 227–243. Kluwer Academic Press, 1992.

[CaGM86] A. Camilleri, M.J.C. Gordon, and Th. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 41–66. North-Holland, 1986.

[Cami88] J. Camilleri. Executing behavioural definitions in higher order logic. Technical Report 140, University of Cambridge Computer Laboratory, 1988.

[Cami91] J. Camilleri. Symbolic compilation and execution of programs by proof: a case study in HOL. Technical Report 240, University of Cambridge Computer Laboratory, 1991.

[Gunt92] E. Gunter. Why we can't have SML-style datatype declarations in HOL. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, volume A-20 of *IFIP Transactions*, pages 561–568, Leuven, Belgium, 1992. North-Holland.

[Hunt86] W.A. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89–132. North-Holland, 1986.

[Jone87] S.L.P. Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[KuSK93] R. Kumar, K. Schneider, and Th. Kropf. Structuring and automating hardware proofs in a higher-order theorem-proving environment. *Journal of Formal Methods in System Design*, 2(2):165–223, 1993.

[Melh88] T. F. Melham. Automating recursive type definitions in higher order logic. Technical Report 146, University of Cambridge Computer Laboratory, September 1988.

[ScKK93c] K. Schneider, R. Kumar, and Th. Kropf. Eliminating higher-order quanitifers to obtain decision procedures for hardware verification. In *International Workshop on Higher-Order Logic Theorem Proving and its Applications*, Vancouver, Canada, August 1993.

[Shee88] M. Sheeran. Retiming and slowdown in Ruby. In G.J. Milne, editor, *The fusion of Hardware Design and Verification*, pages 289–308, Glasgow, 1988. North Holland.

This article was processed using the LaTeX macro package with LLNCS style