Embedding a CHDDL in a Proof System

K. G. W. Goossens

Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, U.K.

Abstract

This paper describes how a formal semantics for a computer hardware design and description language may be embedded in a proof system. An abstraction of ELLA[1], its formal structured operational semantics and the underlying semantic model are introduced. The LAMBDA[2] proof system, the embedding of the semantics and some results are discussed. Some examples are shown, and other approaches are briefly surveyed.

1 Introduction

As circuits are getting larger and more complex, circuit design is becoming more difficult. The need to describe and document designs has led to the development of computer hardware design and description languages (CHDDLs) such as ELLA [Com90] and VHDL [Ins88]. Using these languages, a circuit may be described at all stages of its design, from the high level specification down to the gate level description. Traditionally simulators have been used to test designs at various levels. It is now impossible to fully test a design by simulation alone. Circuits are too large to simulate all possible input combinations. In addition, if a circuit contains internal state the input history must be taken into consideration, increasing the number of possible test cases dramatically. The effectiveness of tools such as simulators must be improved if they are to remain useful. Symbolic simulators such as MOSSYM [Bry85] allow simulation at a higher level, reducing the number of test vectors. Rather than dealing with individual values, variables and formulae, possibly representing more than one value, are used. The major limitation of all conventional simulators is that they can only deal with fixed circuits. We would like to verify general designs such as N bit adders, rather than test each instantiation of a design. This applies not only to parametrised designs, but also to simple components. Rather than re-testing these in every context, it makes more sense to establish the conditions under which they are known to function correctly. Thus verifying a design requires checking functional correctness and validating the contexts in which the components are used. To reason about circuits it is useful to have objects representing them. This allows variables of type circuit, which facilitates a refinement-based approach to design [FM89]. This contrasts with

[1]ELLA is a trademark of the Secretary of State for Defence, United Kingdom.
[2]LAMBDA is a product of Abstract Hardware Limited.

most verification approaches which describe a circuit through its relational or functional behaviour. Meaning must be ascribed to circuit objects if we want to reason about their behaviour. A formal semantics is therefore necessary.

This paper explores the embedding of a fragment of the CHDDL ELLA in the LAMBDA proof system. The next section describes the PICOELLA language we will be working with. A formal semantics is provided in section 3. Section 4 briefly introduces the LAMBDA proof system, before discussing the embedding of PICOELLA in section 5. Section 6 describes some small examples. Related work is discussed in section 7. Finally we discuss future work and conclusions.

2 picoELLA

In order to reason about circuit descriptions in a CHDDL, the CHDDL must have a formal basis. We have written a formal structured operational semantics for PICOELLA, a subset of ELLA encompassing its salient features [Goo90b]. ELLA was chosen because of its elegant data and timing model. Here we describe PICOELLA informally, its formal semantics will appear in the next section.

PICOELLA corresponds to the functional subset of ELLA. Full ELLA may be translated into PICOELLA. It does not contain functions, so that every program is a single expression. Functions may be introduced at a higher level in the proof system. PICOELLA contains the following constructs: type definitions, local declarations, recursive declarations, constants, tuples, indexing, multiplexors and delays. Each of these will be described briefly, followed by an example at the end of this section.

Type definitions are used to define the values that a signal may have. Enumerated types define an arbitrary (finite) number of distinct constructors, such as

```
TYPE bool = true | false IN
     TYPE signal = hi | lo | x | z IN expr
```

Tuple types introduce signals with components made up from previously defined types:

```
TYPE twobool = bool * bool IN expr
```

Local declarations allow signals to be given an explicit name. This aids the structuring of circuit descriptions, and allows fan-out of signals.[3] To be able to deal with feedback, recursive declarations are introduced. For example, the following expression describes an alternating signal `hi`, `lo`, `hi` ... changing at every time tick.

```
LET REC out = DELAY (hi, IF out MATCHES hi THEN lo ELSE hi) IN out
```

A constant is a constructor (from an enumerated type definition), a bottom value, or a tuple containing constants. Associated with every type is a *bottom value* `?type`. It is declared implicitly, and represents the undefined or "don't know" value of that type. The bottom value is used to introduce a *data ordering* $\sqsubseteq$ on constants of the same type. `?type` is less defined than every other value. Constructors are less than or equal to themselves, but incomparable to other constructors. Thus $\sqsubseteq$ is a *flat* data order, and is extended

[3] We will assume throughout this paper that an unlimited fan-out is permitted, although it is possible to limit fan-out.

component-wise to deal with tuples; for example, (hi,?signal) $\sqsubseteq$ (hi,lo). As we shall see in the next section, the ordering $\sqsubseteq$ is crucial to the semantic model.

Tuples and indexing behave as usual; for example ((x,y)[1], (x,y)[2]) is equal to (x,y). Strictly speaking indexing is not required, but it will simplify matters when combining expressions at a meta-level.

Multiplexors are at the basis of programs; they are used to describe a full range of applications, from basic gates to sophisticated features such as bus arbiters [Com86] Section 16.4. For example, an AND gate may be described as follows:

```
IF expr MATCHES (true,true) THEN true ELSE false
```

Another, equivalent, description is

```
IF expr MATCHES (false,bool) | (bool,false) THEN false ELSE true
```

The pattern between the MATCHES and THEN is called a *chooser*. bool is a wild card matching every value of type bool. The bar "|" denotes disjunction: a value matches ch|ch' if it matches one or both of ch and ch'. A tuple represents pairing: it matches if both components match. Only defined values may be used in choosers; one is not allowed to check for the bottom value. The way in which this matching process is defined for bottom values is important. There are three possibilities when matching a value with a chooser: (i) The value and the chooser give a definite *match*. For example, true matches true|false. (ii) The value and the chooser give a definite *no-match*: (hi,lo) does not match (lo,lo). (iii) Finally *neither* a match nor a no-match may occur. Consider either version of the AND gate with input (?bool,true). If the unknown value turned out to be true the output would be true. On the other hand, if it turned out to be false the output would be false. The output of ?bool therefore reflects the intuition that we cannot give a definite answer.

The delay is the final construct in PICOELLA. It introduces a discrete linear time starting at time zero. DELAY(c_t,e) denotes a unit delay with the output of circuit e as its input. DELAY(c_t,e) outputs the result of e one time step after it has been computed. The output at *this* time step is the constant c_t. At time $t+1$ the new value c_{t+1} in the delay is the output of e at time t. The state of the delay (*i.e.* its contents) are explicit in its description.[4] This is in contrast to the more common use of a *state*, which remembers the values of delays from one time step to the next. Embedding the state in the circuit description forces us to evaluate a new program at every time step. The result of an evaluation consists therefore not only of the output signal of the program at that time step but also the description of the program for the next time step.

An example of PICOELLA, which we will refer to throughout this document is a half adder, which delivers its output after one time step:

```
LET s = IF (x,y) MATCHES (true,false)|(false,true) THEN true ELSE false
IN
    LET c = IF (x,y) MATCHES (true,true) THEN true ELSE false
    IN
        DELAY ( ?twobool, (s,c) )
```

The first LET computes the sum, the second the carry. ?twobool is equivalent to (?bool, ?bool). We assume that true, false and twobool have been defined as above.

[4] In ELLA the value in a delay description indicates not the state, but its output *at time zero only*.

3 PicoELLA Semantics

The formal semantics of PICOELLA takes the form of a structured operational semantics [Plo81, Goo90b]. We have written a *static semantics* expressing which programs are well typed for PICOELLA. It will not be discussed in this section. It suffices to say that typing is "intuitively obvious". The *dynamic semantics* evaluates a well typed program in a given environment. There are one or more rules for every construct in the language, expressing their behaviour. There are a total of 18 rules for the 14 constructs in the language. The dynamic semantics acts on a stream of values and a program. The result of a semantic rule at each time step is the new circuit description and a constant. As described in the previous section, rather than using state to remember values in the delays, the circuit description itself is altered. This allows us to use environments (Γ in the rules below) rather than states.

At every time step the element at the head of the input stream is removed from the stream and added to the initial environment. A program has therefore exactly one input and one output. This is not a limitation as these values may be tuples, representing more than one input or output signal. For a completed program the initial environment will be empty, but partial programs may have "unconnected wires". These may be set to a constant value during simulations by providing a non empty initial environment. This is an important feature, as it allows us to reason about program fragments in general contexts. We could show for example, that a given circuit functions correctly using several timing disciplines.

Let the initial program and input stream be e_0 and $< i_0, i_1, \ldots, i_N >$ respectively. The result of an evaluation of program e_t at time t with input i_t consists of a tuple (o_t, e_{t+1}) representing the output signal of the circuit, and a description of the circuit at the next time step $t+1$. Time is explicit only in three rules dealing with input; the remainder of the dynamic semantics is independent of time. The rule for `LET` expresses that *expr* is evaluated in the environment Γ' resulting from evaluating the declaration *decl*. The result of the `LET` consists of the output of *expr* at the current time, and the circuit description at the next time step. This new circuit is a `LET` built up from the new declaration *decl'* and the new expression *expr'*.

$$\frac{\Gamma \vdash \ decl \Rightarrow \ \Gamma',\ decl' \qquad \Gamma' \vdash \ expr \Rightarrow \ v,\ expr'}{\Gamma \vdash \ \texttt{LET}\ decl\ \texttt{IN}\ expr \Rightarrow \ v,\ \texttt{LET}\ decl'\ \texttt{IN}\ expr'} \tag{1}$$

The following rule shows how a non recursive declaration is evaluated.

$$\frac{\Gamma \vdash \ expr \Rightarrow \ v,\ expr'}{\Gamma \vdash \ name = expr \Rightarrow \ \Gamma\{(name, v)\},\ name = expr'} \tag{2}$$

$\Gamma\{(name, v)\}$ is the environment which contains value v for name *name*, and is identical to Γ otherwise. The next rule shows clearly that the output of the delay is the constant value contained within it. The new delay contains the output v from *expr* at the current time step. The bijective function *valueof: Expr* $\rightarrow$ *Value* converts a constant expression to the corresponding constant value.

$$\frac{\Gamma \vdash \ expr \Rightarrow \ v,\ expr'}{\Gamma \vdash \ \texttt{DELAY}\ (c,\ expr) \Rightarrow \ valueof(c),\ \texttt{DELAY}\ (valueof^{-1}(v),\ expr')} \tag{3}$$

The following rule will clarify the informal description of the multiplexor, or `IF` statement. Firstly, this rule evaluates all sub-expressions of the `IF`. It is *strict* because every branch, whether its answer is used or not, must compute its circuit description for the next time step. The three cases outlined previously are represented in Kleene's ternary logic as *tt* (a match), *ff* (a no-match) and $\bot_{Bool}$ (neither a match nor a no-match).

$$\frac{\Gamma \vdash \ expr_0 \Rightarrow \ v_0, \ expr_0' \qquad \Gamma \vdash \ expr_1 \Rightarrow \ v_1, \ expr_1' \qquad \Gamma \vdash \ expr_2 \Rightarrow \ v_2, \ expr_2'}{\begin{array}{l}\Gamma \vdash \ \texttt{IF}\ expr_0\ \texttt{MATCHES}\ chooser\ \texttt{THEN}\ expr_1\ \texttt{ELSE}\ expr_2 \\ \Rightarrow \ v, \ \texttt{IF}\ expr_0'\ \texttt{MATCHES}\ chooser\ \texttt{THEN}\ expr_1'\ \texttt{ELSE}\ expr_2'\end{array}} \tag{4}$$

Where *type* is the type of $expr_1$ and $expr_2$ in:

$$v \equiv \begin{cases} v_1 & match(v_0, \ chooser) = tt \\ v_2 & match(v_0, \ chooser) = ff \\ ?type & match(v_0, \ chooser) = \bot_{Bool} \end{cases}$$

We may prove that the *match* function is *monotone* in its second argument, *i.e.* if the value of an input becomes more defined then the output becomes more defined. To be more precise:

$$\vdash \forall ch, c, d.\ c \sqsubseteq d \rightarrow match\ ch\ c \sqsubseteq_3 match\ ch\ d \tag{5}$$

$\sqsubseteq$ is the data ordering defined in the previous section. $\sqsubseteq_3$ is a similar data ordering on *tt*, *ff* and $\bot_{Bool}$. The reason for forbidding the use of bottom values `?type` in choosers is that it would allow non-monotone circuit descriptions.

The `LET REC` operator is used to describe feedback, which means that we have to use the output value *before* we have computed it. The dynamic semantics computes the *least fixed point* of the circuit output. Using an iterative method, this guarantees that every circuit will return a sensible answer in a finite number of steps. This is particularly important if a circuit could oscillate. Consider the alternating circuit of page 2 *without* the delay:

```
LET REC out = IF out MATCHES hi THEN lo ELSE hi IN out
```

If the value on the `out` wire happened to be `hi` the inverter would output `lo`. This means that its input changes to `lo`. Hence its output changes back to `hi`. This circuit is therefore unstable for output `hi` and `lo`.[5] However, the circuit is stable for `?signal`: it is the least fixed point of this circuit. It is fixed, *i.e.* it does not oscillate, and it is the least because `?signal` is the smallest value of type `signal`.

To prove that the dynamic semantics computes the least fixed point we use the Knaster-Tarski theorem. The theorem states that the least fixed point of a continuous function f for some complete partial order is equal to the least upper bound of the set of all iterative applications of f to the bottom element $\bot$ of the CPO, *i.e.* the set $\{\bot, f(\bot), f(f(\bot)), \ldots\}$. It has been shown that the the data ordering $\sqsubseteq$ is a CPO, and that the semantics is monotone and continuous.

As an example consider the half adder

[5] Actually, it is also unstable for the remaining constructors `x` and `z`.

```
LET s = IF (x,y) MATCHES (true,false)|(false,true) THEN true ELSE false
IN
    LET c = IF (x,y) MATCHES (true,true) THEN true ELSE false
    IN
        DELAY ( ?twobool, (s,c) )
```

Some indication how this circuit behaves is shown in the following table. Note that the value in the delay is the value computed by the XOR and AND circuits at the preceding time step.

time	input	program	output
0	`(true,false)`	`LET ... DELAY (?twobool,(s,c))`	`(?bool,?bool)`
1	`(true,true)`	`LET ... DELAY ((true,false),(s,c))`	`(true,false)`
2	`(false,false)`	`LET ... DELAY ((false,true),(s,c))`	`(false,true)`
3	...	`LET ... DELAY ((false,false),(s,c))`	`(false,false)`

4 Lambda

This section briefly describes the LAMBDA proof system [FFM90], and its relevant features. The LAMBDA proof system is an implementation of a polymorphic constructive higher order logic of partial terms. Polymorphism allows functions to operate on objects of more than one type. The logic is constructive, which implies that we do not have the law of the excluded middle or a strong axiom of choice. This contrasts with a classical system such as HOL, in which Hilbert's ε operator is used for descriptions. Higher order logic means that we can reason about functions, functions of functions *etc.* For example, signals are usually represented as functions from time to values, and circuits are described in terms of their effect on signals. To reason about partial terms an existence predicate `E`, weak equality or equivalence `===`, and strong equality `==` are included. The system is implemented in ML which is also used as the command language [HMT89].

LAMBDA allows the user to declare new data types and functions, which may then be used in rules, theorems *etc.* LAMBDA's data type and function definitions are a large subset of standard ML. For example,

```
datatype bool = true | false; fun not true = false | not false = true;
```

is part of the built-in definitions of the `bool` data type. LAMBDA returns a number of rules and theorems which axiomatise this data type and function. For example, rules about a data type are: existence of constructors, (in)equality rules for the constructors and an induction rule for the type.

```
****** bool'ind ******
2:  G // H |- Pbool#(false)
1:  G // H |- Pbool#(true)
    --------------------
    E w $ G // H |- Pbool#(w)
```

This induction rule allows us to derive things like $\forall x : bool.\ x == true \vee x == false$. For functions, rules include existence of the function and its partial applications, and rules which allow rewriting. `not true` is equivalent to `false` for example:

```
****** not'eq'1 ******
   --------------------
   G // H |- not true === false
```

In addition to rules, LAMBDA has tactics. Using tactics a number of rules may be applied in succession / in any order / repeatedly / only if applicable *etc.* A sophisticated rewrite system is also available. A number of libraries with rules and tactics about lists, booleans, words, naturals and integers are supplied with LAMBDA.

5 Embedding picoELLA in Lambda

This section describes how the greater part of the static and dynamic semantics of PICOELLA has been embedded in the LAMBDA proof system. We will encode the types and functions used in the semantics using the ML definitions mechanism explained in the previous section.

The first thing we need are constants:

```
datatype const = Cons of natural * natural | CoTuple of const * const;
```

`Cons(i,t)` encodes the i^{th} constructor of type `t`. By convention `Cons(0,t)` represents `?t`. A constant is therefore a constructor or bottom value, or a tuple containing constants. The structural induction rule the system returns for constants is:

```
2:  E r1' $ E r' $ G // P#(r1') $ P#(r') $ H |- P#(CoTuple (r1',r'))
1:  E r3' $ E r2' $ G // H |- P#(Cons (r3',r2'))
    --------------------
    E w $ G // H |- P#(w)
```

To prove a property `P` of all constants `w` two subgoals must be proved: (i) assuming that naturals `r3'` and `r2'` exist, prove the base case `Cons`; (ii) assuming that `P` holds for constants `r1'` and `r'`, prove the inductive step for `CoTuple`. In this rule, the hypotheses left of the `G` (known as the G-list) are conventionally *existence* hypotheses. Those between the slashes and `H` are the remaining hypotheses.

To encode the matching process for the `IF` construct we define

```
datatype choosers = C of const
                  | B of choosers * choosers
                  | T of choosers * choosers;
fun match (C c) c' = ...|
    match (B (b,c)) a = or3 (match b a) (match c a) |
    match (T (a', b')) (CoTuple (a,b)) = and3 (match a' a) (match b' b);
```

Thus `choosers` are either constants, bars, or tuples. As described on page 3, the bar corresponds to disjunction, and tuples to pairing. `and3` and `or3` are part of a general purpose three valued boolean logic library, specifically written for this project. The description for matching one constant with another has been omitted here, because it would obscure the definition. Note that `match` is a *partial* function; the result of `match (T ...) (Cons ...)` is not defined. The function is total in the case where the types of the chooser and constant are equal. The definitions for `typeOfChoosers`, `typeOfConst`, and the type `tpe` representing types are straightforward. We are now in a position to re-prove the monotonicity of `match` (equation 5), but now within LAMBDA:

```
G // H |- forall ch,c,d.  le c d == true /\
                          typeOfChoosers ch == (typeOfConst c, true) /\
                          typeOfChoosers ch == (typeOfConst d, true) ->>
                          le3 (match ch c) (match ch d) == true
```

The type of circuits is defined as follows:

```
datatype expr = Const of const
              | Tuple of expr * expr
              | Let of expr * expr
              | Var of natural
              | Delay of const * expr
              | If of expr * expr * expr * choosers
              | Index1 of expr
              | Index2 of expr;
```

Note that the `LET REC` and `TYPE` constructors are absent. Work is in progress to add the `LET REC` operator. To embed the `LET` operator a de Bruijn encoding of lambda abstractions was used [dB72]. The bound variables of lambda expressions are encoded as natural numbers indicating the distance (measured in intervening lambdas) away from the defining lambda. Thus $\lambda x.\lambda y.(x,(x,y))\ a\ b$ would be encoded as $\lambda\lambda(1,(1,0))\ a\ b$. In PICOELLA this corresponds to encoding `LET x = a IN LET y = b IN (x,(x,y))` by `Let (a, Let (b, Tuple (Var 1, Tuple (Var 1, Var 0))))`. This encoding was necessary because a formal description of `names` of PICOELLA had to be found in LAMBDA.

Finally, the dynamic semantics can be defined:

```
fun Reduce l (Delay (c,e)) = (c, Delay (Reduce l e)) |
    Reduce l (Let (e,e')) =
           let val (c, f) = Reduce l e
               val (c', f') = Reduce (c::l) e'
           in
               (c', Let (f,f'))
           end |
    Reduce l (Var n) = elem l n |
    Reduce l (If (e,e',e'',ch)) =
           let val (c,d) = Reduce l e
               val (c',d') = Reduce l e'
               val (c'',d'') = Reduce l e''
           in
               ( case match ch c of
                 tt => c' |
                 ff => c'' |
                 uu => bottom c',
                 If (d,d',d'',ch) )
           end | ...;
```

This definition reflects the semantic rules at page 4. Consider the clause for a delay. The output at this time step is the value `c`, which was stored in the delay. The new circuit is

Delay (c',e') where c' is the output from the circuit e at this time step, and e' is the description of circuit e at the next time step.

The LET statement reduces the defining expression, and pushes the value result on the stack (*i.e.* stores it in the environment). Evaluating a name corresponds to a lookup in the environment Γ in the dynamic semantics, and a lookup in the stack l in the embedding.

The IF construct evaluates all of its sub-expressions. It then returns (i) the result of the first branch (if we have a definite match — *tt* or tt); or (ii) the result of the second branch (if we have a definite no-match — *ff* or ff); or (iii) a bottom value of the appropriate type (if we have neither a definite match nor a definite no-match, $\bot_{Bool}$ or uu). As in rule (4), the IF is strict.

We may now prove properties which we would like to hold for the semantics. One such property is the monotonicity of the reduction function. By extending the data ordering on constants $\sqsubseteq$ to lists $\sqsubseteq_l$, the following theorem has been proved:

$$\vdash \forall l, l', e.\ \mathit{welltyped} \wedge l \sqsubseteq_l l' \to\ \exists c, f, c', f'.\ \ \begin{array}{l}\texttt{Reduce}\ l\ e == (c, f) \wedge \\ \texttt{Reduce}\ l'\ e == (c', f')\ \wedge c \sqsubseteq c'\end{array} \tag{6}$$

welltyped stands for a number of properties, which we will see below. Consider the program consisting of a single delay: DELAY(?bool,Var 0). If l == [?bool] and l' == [false], then (6) tells us that ?bool $\sqsubseteq$?bool. At the next time step, however, we cannot use the theorem because we now have two *distinct* circuits (DELAY(?bool,Var 0) and DELAY(false,Var 0) respectively). We introduce an ordering on programs $\sqsubseteq_p$, and a predicate $=_{shape}$ determining when two circuits have "the same shape". $\mathtt{e} =_{shape} \mathtt{e}' \wedge \mathtt{e} \sqsubseteq_p \mathtt{e}'$ indicates that e and e' are identical, except that constants in e are less than or equal to those in e'. Thus DELAY(?bool,Var 0) $\sqsubseteq_p$ DELAY(false,Var 0). Now we can state a more general monotonicity theorem:

```
G // H |- forall l,l',e,e',t.
                 lle l l' == true /\
                 ple e e' == true /\
                 shapeEq e e' == true /\
                 (map typeOfConst l) == (map typeOfConst l') /\
                 typeOfExpr (map typeOfConst l) e == (t, true) ->>
                 exists c,f,c',f'.
                        Reduce l e == (c, f) /\
                        Reduce l' e' == (c', f') /\
                        le c c' == true /\
                        ple f f' == true /\
                        shapeEq f f' == true /\
                        shapeEq e f == true /\
                        typeOfConst c == t /\
                        typeOfConst c' == t
```

It states that for all environments l and l', circuit expressions e and e', and types t, if $\mathtt{l} \sqsubseteq_l \mathtt{l}' \wedge \mathtt{e} \sqsubseteq_p \mathtt{e}' \wedge \mathtt{e} =_{shape} \mathtt{e}'$ and environments l and l' have the same type and circuit e is well formed in l interpreted as a type environment, the following properties hold. (i) Reduce l e and Reduce l e' exist (*i.e.* the semantics terminates within a finite number of steps) and are equal to (c,f) and (c',f') respectively; (ii) Reduce is monotone in its first and second arguments; and (iii) `Reduce` preserves shape equality, well formedness and types. In other words, if we start with any well typed circuit and evaluate it in a number of different environments then all the resulting circuits will have the same shape, will be well typed and will be ordered as the environments.

6 Worked Examples

This section shows how LAMBDA may be used to manipulate circuits. We show how applications of a tactic compute the normal form of PICOELLA circuit expressions. In other words, a symbolic simulation is performed. The example also illustrates the use of LAMBDA's abbreviations and functions to augment PICOELLA.

Consider the following definitions:

```
val unknown = Cons(0,1);
val hi      = Cons(1,1);
val lo      = Cons(2,1);
val OR#(x)  = If (x, Const lo, Const hi, T (C lo, C lo));
val AND#(x) = If (x, Const hi, Const lo, T (C hi, C hi));
val XOR#(x) = If (x, Const lo, Const hi,
                  B (T (C hi, C hi), T (C lo, C lo)));
val HA#(x) = Let (x, Tuple (XOR#(Var 0), AND#(Var 0)));
fun adder x y c = Let (Tuple (Tuple (x, y), c),
                  Let (HA#(Tuple (Index1 (Index1 (Var 0)),      (*x*)
                                  Index2 (Index1 (Var 0)))),    (*y*)
                  Let (HA#(Tuple (Index1 (Var 0), Index2 (Var 1)(*c*))),
                       Tuple ( Index1 (Var 0),
                               OR#(Tuple (Index2 (Var 1),
                                          Index2 (Var 0)))))));
```

`hi`, `lo`, `AND` *etc.* are *abbreviations* which may or may not have arguments. Abbreviations are constant syntactic functions with no free variables. The full-adder `adder` uses a function instead of an abbreviation. Note that we are using LAMBDA's definition mechanism to structure PICOELLA terms. These definitions may be loaded into LAMBDA using

```
val ( pe',r' ) = useLambda pe "definitions";
```

`pe'` is a new parser environment, `r'` are the rules characterising the definitions. Parser environments ensure that new definitions are known to the parser, and are printed out correctly.

The half adder `HA` has one argument, which is a tuple. It may be interpreted as a pair of input wires. Similarly its output is a tuple consisting of a (sum,carry) pair. Any function using the half adder, must select the appropriate components of the half adder's result. The `Let` is used to explicitly name all inputs to the function (which may be large expressions) so that these are computed only once, after which the result is distributed by using a fan-out. In the case of the full-adder, the `Let` is necessary to ensure the correct offset for any `Var i` inputs. `x` and `y` are added first, followed by the addition of the partial sum and `c`. The final result is the disjunction of the two carries (the second components of the half adders), and the sum of the second component. Without the indexing operators it would not have been possible to partition this description into subcomponents. As described in section 3, "unconnected wires" are represented by accesses to the environment. This can be expressed in the following way:

```
Reduce (hi::  hi::  lo::  l) adder (Var 0) (Var 1) (Var 2) ==
                              (output, adder (Var 0) (Var 1) (Var 2))
```

The adder instantiation uses the top three elements of the environment, with the `Var i` arguments. The remaining environment `l` is irrelevant. If we use the adder in a larger circuit, it may be used in different contexts (*i.e.* `l` will be instantiated with different environments) as long as the top three components are there (and are of the right type). This term also asserts that the adder does not change over time, and therefore has no internal state. The circuit description for the next time step is `adder (Var 0) (Var 1) (Var 2)`, which is also the current circuit.

We may now prove properties about these circuits, for example:

```
***** Level 1  Premise 1 *****
1:  x == hi \/ x == lo \/ x == unknown
2:  input == CoTuple (x,lo)
3:  halfadder == HA#(Var 0)
|- Reduce [input] halfadder == (CoTuple (sum,carry),next_halfadder)
--------------------
G // x == hi \/ x == lo \/ x == unknown $
input == CoTuple (x,lo) $ halfadder == HA#(Var 0) $ H
|- Reduce [input] halfadder == (CoTuple (sum,carry),next_halfadder)
```

After removing the abbreviations we apply the tactic `ReduceAllTac`, which reduces expressions involving `Reduce` to a normal form. After the re-introduction of abbreviations the result is:

```
***** Level 8  Premise 1 *****
1:  input == CoTuple (x,lo)
2:  halfadder == HA#(Var 0)
|- sum == x /\ carry == lo /\ next_halfadder == halfadder
--------------------
G // x == hi \/ x == lo \/ x == unknown $
input == CoTuple (x,lo) $ halfadder == HA#(Var 0) $ H
|- Reduce [input] halfadder == (CoTuple (sum,carry),next_halfadder)
```

This proof takes seven steps, each of which was a straightforward application of a tactic. Note that we have not just simulated the circuit, we have established that with a `lo` carry input, the sum is identical to the other input, even for the "don't know" value.

A specification for the half adder circuit could be stated as follows

```
val HA_SPEC#(x,y,sum,car) = sum == (x+y) mod 2 /\ car == (x+y) div 2;
fun abs (Cons(1,1)) (* hi *) = 1 | abs (Cons(2,1)) (* lo *) = 0;
```

Using the abstraction function `abs`, we may then prove that the half adder implementation satisfies the specification. Alternatively, we may design a circuit using refinement by starting with the rule:[6]

[6] Some typing constraints have been left out for clarity.

```
***** Level 1  Premise 1 *****
E impl $ E x $ E y $ E s $ E c $ G //
Reduce [CoTuple(x,y)] impl == (CoTuple(s,c),impl) $ H
|- HA_SPEC#(abs x, abs y, abs s, abs c)
--------------------
E impl $ E x $ E y $ E s $ E c $ G //
Reduce [CoTuple(x,y)] impl == (CoTuple(s,c),impl) $ H
|- HA_SPEC#(abs x, abs y, abs s, abs c)
```

We may then split `impl` into two circuits, one to compute the sum and one to compute the carry.

```
***** Level n  Premise 1 *****
E sumc $ E x $ E y $ E s $ G //
Reduce [CoTuple(x,y)] sumc == (s,sumc) $ H
|- (abs x)+(abs y) mod 2 == abs s
**** Level n  Premise 2 *****
E carc $ E x $ E y $ E c $ G //
Reduce [CoTuple(x,y)] carc == (c,carc) $ H
|- (abs x)+(abs y) div 2 == abs c
--------------------
E (Tuple(sumc,carc)) $ E x $ E y $ E s $ E c $ G //
Reduce [CoTuple(x,y)] (Tuple(sumc,carc)) ==
(CoTuple(s,c),(Tuple(sumc,carc)) ) $ H
|- HA_SPEC#(abs x, abs y, abs s, abs c)
```

This process may then be repeated for the `sumc` and `carc` subcircuits until a complete implementation has been found.

To show the power of having explicit circuit descriptions consider the following (simplified) *hardware generating* function.

```
fun nadder 1 x y c = adder x y c |
    nadder (S (S n)) (Tuple (x,x')) (Tuple (y,y')) c =
           Let (nadder (S n) x' y' c,
           Let (adder x y (Index2 (Var 0)),
                Tuple (Tuple (Index1 (Var 0), Index1 (Var 1)),
                       Index2 (Var 0))));
```

It returns a description for an N bit adder for arbitrary $N > 0$. A one bit adder is the full-adder described above. An $N+1$ bit adder is an N bit adder followed by a full-adder which adds the most significant bits and the carry from the N bit adder. The final result is a nested tuple

```
Tuple ( Tuple (msb, Tuple (..., lsb)...), carry)
```

where `msb` and `lsb` are the most and least significant bits respectively.

7 Related Work

In [BGM91] Barringer *et al.* describe a language akin to PICOELLA. Their Logic+Delay language is also an abstraction from ELLA. The data values are either `true` or `false`. There are no tuples. Their `CASE` statement corresponds to PICOELLA's `IF expr MATCHES true THEN expr' ELSE expr''`. The language also has a stable delay of arbitrary length. Rather than including explicit declarations, a program is a set of definitions, which may contains loops. Thus fan-out and feedback loops are permitted.

The semantic model uses (finite) histories to save the internal state of delays. At every clock tick all declarations are evaluated simultaneously until a stable solution has been found. An interesting difference in the treatment of undefined values comes to light in one of the delay rules. It uses a *negative judgement* $\not\vdash$ to detect unstable signals:

$$\frac{h \not\vdash_t rhs \Rightarrow_\triangle 0 \qquad h \not\vdash_t rhs \Rightarrow_\triangle 1}{h \vdash_t rhs \Rightarrow_\triangle ?} \tag{7}$$

This rule expresses that if rhs does not evaluate to either 0 or 1 using history h, it evaluates to ? (`?bool`). Undefined elements can arise either from a delay whose input has not remained stable for a sufficiently long interval, or from an unstable loop (*c.f.* the alternating circuit on page 5). A rule similar to the one just discussed detects unstable circuits. Informally, it states that the output of the circuit is undefined due to instability if after the circuit reaches a state h'' from the initial state h', there exists a non empty sequence of states $h'', \ldots, h''$. That is, the evaluation is stuck in a loop. Both of these rules could be embedded in a way similar to PICOELLA. The second rule would have to keep track of all previous states, and check for a circularity at each step. It is not clear however, how to use these rules within a constructive proof system.

In [BH89] Brock and Hunt use the Boyer-Moore theorem prover to encode a circuit type. The Boyer-Moore system implements a quantifier free first order predicate logic with equality. Circuits are encoded as constants in the logic, and are given a meaning using interpreters. A well formedness predicate is used to recognise valid circuit descriptions. Circuit descriptions resemble Lisp. Brock and Hunt demonstrate circuit generating functions such as N bit adders which are also verified. A severe drawback to their approach is that circuit descriptions are restricted to combinatorial circuits with no feedback. This allows the interpreters and well formedness predicates to be relatively simple. The complexity of the PICOELLA semantics is due solely to these features.

Boulton *et al.* have implemented a *behaviour extraction function* [BGHvT90] in the HOL proof system. It maps ELLA descriptions from outside the proof system onto (classical) higher order logic formulae. A large functional subset of ELLA, including functions, is thereby given a semantics. However, the lack of a circuit type prevents results about general or partially instantiated circuits to be proved.

A behaviour extraction function maps a circuit onto a formula describing its meaning. A problem with general behaviour functions is that they do not reside in the proof system, preventing a formal verification. The Cambridge system, however, counters this problem by making the result of the behaviour function as close to the original ELLA as possible. The model within HOL is at the same level of abstraction as the original ELLA. Ideally this high level embedded semantics would correspond to a formal semantics outside the proof system, as is the case for PICOELLA. [Goo90a] contains a formal semantics for a superset of the ELLA subset used by Boulton. Whether this semantics agrees with their embedded semantics is not clear.

In contrast, an embedded operational semantics can be reasoned about formally. We can thus gain confidence in the semantics by formally proving desirable properties. For PICOELLA, the correspondence between "paper" and embedded semantics is very close, again increasing our confidence in the correctness of the embedding (with respect to the "paper" semantics). PICOELLA's semantics works at a lower level, by explicitly embedding the fix point model. A fine grained semantic model allows more detailed results to be proved about the language. Also, by embedding a much smaller subset of ELLA, it is easier to maintain a consistent system. The remainder of the language can later be given a semantics in terms of the embedded subset. ELLA descriptions outside the proof system may then be mapped onto syntactically equivalent proof system term, which have a formal semantics by virtue of the embedded operational semantics.

8 Future Work and Conclusions

Two short terms goals are the embedding of the `LET REC` construct, and the completion of a number of normal form results for choosers. Proving the correctness of the fix point result will be challenging. The implementation of a pretty printing system, to present PICOELLA expressions in a more readable form, will make the system more user friendly. A number of tactics and rules to perform common operations with a minimum of effort would be very useful. In addition the semantics should be integrated with LAMBDA's window based browser facility.

As mentioned in the previous section the remainder of ELLA can, in principle, be given a derived semantics. This may be achieved by encoding a second circuit type encompassing the new constructs, and then formally mapping these constructs onto PICOELLA. Also, a behaviour function could be derived by proving a characterisation of every construct. An informal function can then map every construct onto the relevant property.

The embedding of the state of delays in the circuit description simplifies the semantics, but complicates reasoning about general circuits. It would be very helpful if a general method for dealing with delays could be found. Combining PICOELLA with various design strategies, such as design for correctness, formal system design [FM89] and transformational design [Bus90] may be possible. Hardware generating functions, such as the N bit adder generator, open up the exciting prospect of formally verified hardware generators.

The efficiency aspects of the operational semantics will have to be addressed at some point. Brute force methods do not work well even on the small examples shown above. An incremental approach to verification has so far been faster. In addition, it allows the re-use and structuring of designs and proofs, which will be crucial for circuits of medium to large size. Finally, a number of larger case studies need to be performed to see how the design activity is improved, speeded up, or otherwise.

Formal semantics for CHDDLs will be increasingly important as the need to verify circuit descriptions and designs will become increasingly important. This research is a first step towards providing systems which will allow designers to create fully verified circuits. These systems may be hybrid simulator-theorem provers or totally formal. So far very little work has been done about either formalising CHDDLs or the use of CHDDLs in conjunction with, or within proof systems.

References

[BGHvT90] Richard Boulton, Mike Gordon, John Herbert, and John van Tassel. The HOL verification of ELLA designs. Technical Report 199, University of Cambridge Computer Laboratory, August 1990.

[BGM91] Howard Barringer, Graham Gough, and Brian Monahan. Operational semantics for hardware design languages. Technical Report Series UMCS-91-2-2, Department of Computer Science, University of Manchester, February 1991.

[BH89] Bishop C Brock and Warren A Hunt Jr. The formalization of a simple hardware description language. In Luc Claessen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 778–792, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers B.V.

[Bry85] Randal E Bryant. Symbolic verification of MOS circuits. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 419–438, March 1985.

[Bus90] Holger Busch. Proof-based transformation of formal hardware models. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*, Oxford, September 1990.

[Com86] Computer General Electronic Design, The New Church, Henry St, Bath BA1 1JR, England. *The ELLA Tutorial*, issue 3.0 edition, 1986.

[Com90] Computer General Electronic Design, The New Church, Henry St, Bath BA1 1JR, England. *The ELLA Language Reference Manual*, issue 4.0 edition, 1990.

[dB72] N D de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag Math.*, 34:381–392, 1972.

[FFM90] Mick Francis, Simon Finn, and Ellie Mayger. *Reference Manual for the Lambda System*. Abstract Hardware Limited, version 3.2 edition, November 1990.

[FM89] Michael P Fourman and Eleanor M Mayger. Formally based system design – interactive hardware scheduling. In G Musgrave and U Lauther, editors, *International Conference on VLSI*, Munich, 1989.

[Goo90a] K G W Goossens. An operational semantics for a subset of the HDDL ELLA. Version 0.3 Manuscript, April 1990.

[Goo90b] K G W Goossens. Semantics for picoELLA. Manuscript, June 1990.

[HMT89] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ML version 3. LFCS Report Series ECS-LFCS-89-81, LFCS, Department of Computer Science, University of Edinburgh, May 1989.

[Ins88] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY10017 USA. *IEEE Standard VHDL Language Reference Manual*, IEEE std 1076-1987 edition, 1988.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University (DAIMI), 1981.