Embedding Hardware Description Languages in Proof Systems

Kees G. W. Goossens

Doctor of Philosophy

University of Edinburgh

1992

Abstract

The aim of this thesis is to investigate the integration of hardware description languages (HDLs) and automated proof systems.

Simulation of circuit designs written in an HDL is an important method of testing their correctness. However, due to the combinatorial explosion of possible inputs it is not feasible to verify designs using simulation alone. Formal hardware verification, using a proof system, has tried to address this issue. Whilst some medium-sized designs have been (partially) verified, industrial take-up of formal methods has been slow. This is partly due to the use of specialised, non-standard notations employed in various formalisms.

By embedding a hardware description language in a proof system we hope to clarify the semantics of the particular HDL, and present a more standard interface to formal methodologies. We have given a new static structural operational semantics for a subset of the ELLA hardware description language. The formal dynamic semantics of this subset is based on an existing informal model.

We embedded the semantics of this HDL in the LAMBDA higher-order logic proof system. The embedding allows meta-theoretical results to be proved about this and other semantics. It has been proved that the semantics computes the least fixed point solution of the circuit description. Another semantics which computes a more defined output has also been embedded, and the relationship between both semantics has been proved formally.

A number of paradigms such as operational semantics based formal symbolic simulation, formal interactive (top-down and bottom-up) synthesis, formal hardware generators, proved correct transformations and traditional hardware verification are presented as small case studies. However, scaling up of the examples turned out to be difficult, and verification tended to be slow.

Acknowledgements

I would like to thank first of all my supervisor, Stuart Anderson, for his encouragement and hints in the development of this thesis. I would also like to thank my second supervisor Professor Mike Fourman for providing financial support for a substantial period of this research.

The technical support staff of the Department of Computer Science at Edinburgh provided a much appreciated smooth running computing environment.

Mike Hill, James McKinna, Matthew Morley, David Rees and Fabio da Silva proof-read parts of this thesis. My thanks go to them. Any remaining mistakes are, of course, entirely ~~their~~ my responsibility.

Special thanks to Fabio and Hans for being such good office-mates.

Declaration

This thesis has been composed by myself. The work reported herein has not been presented for any university degree before, and, unless otherwise stated, is my own.

Kees G. W. Goossens

Table of Contents

List of Figures

Huius farinae sunt et isti, qui libris edendis famam immortalem aucupantur. Hi cum omnes mihi plurimum debent, tum praecipue ii, qui meras nugas chartis illinunt. Nam qui erudite ad paucorum doctorum iudicium scribunt, quique nec Persium nec Laelium iudicem recusant, mihi quidem miserandi magis, quam beati videntur, ut qui sese perpetuo torqueant: addunt, mutant, adimunt, reponunt, recudunt, ostendunt, nonum in annum premunt, nec umquam satisfaciunt ac futile praemium, nempe laudem, eamque perpaucorum, tanti emunt, tot vigiliis, somnique, rerum omnium dulcissimi, tanta iactura, tot sudoribus, tot crucibus. Adde nunc valetudinis dispendium, formae perniciem, lipitudinem aut etima caecitatem, paupertatem invidiam, voluptatum abstinentiam, senectutem praepoperam, mortem praematuram, et si qua sunt alia eiusmodi. Tantis malis sapiens ille redimendum existimat, ut ab uno aut altero lippo probetur.

Erasmi Roterodami
ΜΩΡΙΑΣ ΕΓΚΩΜΙΟΝ
Id Est STULTITIAE LAUS

In Praise of Folly [53, pp 202–204]

Chapter 1

Introduction

1.1 Hardware Description Languages and Simulation

Modern hardware designs are complex. Conventional methods take many iterations to arrive at an acceptable implementation. Increasing use of circuits in embedded systems, perhaps with some aspect of safety criticality, requires greater rigour in specification and design.

Traditionally, breadboarding has been used to test designs. Breadboarding is the process of constructing the design and then using this prototype to perform tests. However, this is only feasible for small designs. With greater integration and density of components this method becomes prohibitively expensive.

Greater circuit complexity encouraged the use of structured circuit descriptions, leading to hardware description languages (HDLs) [100]. A large number of HDLs have been designed; well-known HDLs include DDL [50], the CONLAN family [147,148][1], ELLA[2] [45] and VHDL [101]. Since their initial use as documentation and design tools, HDLs have been used to *simulate* the design. A simulator for

[1] Citations are ordered by date, and are therefore not necessarily in ascending order.

[2] ELLA is a trademark of the Secretary of State for Defence, United Kingdom.

an HDL is intended to model the behaviour of a circuit described in the language. Given a set of input values, corresponding outputs are computed using a particular simulation model. There is a wide spectrum of such languages, from the very low-level (*e.g.* SPICE [126] which models hardware at the differential-equation level), to high-level languages such as ELLA and VHDL, which contain conventional programming language constructs. Traditionally, simulators have been used to test designs at various levels of abstraction. However, a serious problem of both breadboarding and value simulation is that for any substantial circuit the number of possible inputs (or *test vectors*) becomes very large. If, in addition, a circuit contains internal state the input history must be taken into consideration, increasing the number of possible test cases dramatically [24].

It is not feasible to verify current circuit designs using exhaustive simulation alone. By introducing extra values in the value domain, such as 'don't know' and 'don't care,' the number of test vectors may be reduced substantially [170]. If a particular input is irrelevant for a particular test, its value can be set to don't care, instead of having to simulate the test twice, with the value set to true and false respectively. A number of methods to extend the basic value domain are described in [91].

The next development was to adapt powerful symbolic execution of conventional programming languages to the domain of hardware description [37]. Symbolic simulators such as MOSSYM [25] and later versions of COSMOS [27] allow variables and formulae as inputs. Variables and formulae, possibly representing more than one value, are used instead of individual values. This can drastically reduce the number of test vectors needed to exhaustively test a design. Variables range over all values in the value domain, in contrast to values such as don't know, which are constants in the value domain. Of course, this puts an extra burden on the simulator which now needs to be able to handle arbitrary formulae instead of simple values. It may also require algebraic capabilities to simplify intermediate formulae. In theory, we need to do only one simulation, namely the one with all the input values set to variables. The result would be an expression which would describe the circuit's behaviour. However, this expression may be as complex as the circuit description and simplification of intermediate formulae may be very complex (NP-hard in the case of MOSSYM [25].)

In MOSSYM we have an asymmetry: we are permitted abstraction over data but not over circuits. In other words, we may have symbolic variables ranging over data values, but we are not allowed circuits containing symbolic variables. Symbolic variables are abstract hardware. This idea is not as strange as it may seem; plug-in components are in effect abstract hardware, certainly as long as the circuit is under development. Why would it be useful to have this capability? It would seem that, since we are dealing with the design of a certain circuit, we would only want to simulate that circuit. Consider, however, that large circuits are designed in a modular fashion to allow a number of people to work on separate parts of a circuit at the same time. When a subcomponent is ready, it has to be simulated in a larger context, all of which may not be completed. The availability of an abstract implementation for unfinished parts of the design would enable the component to be simulated in its correct context. A suitable simulator would allow the evaluation of a circuit containing a mixture of concrete and abstract components. Of course, certain properties of the abstract components may be needed to arrive at an output, but these should be available from their specifications. We use the specification of the abstract components to simulate them as long as they are not available. Simulators of multi-level HDLs, which span more than one level of abstraction, often allow a basic form of this. By providing a behavioural description, without any indication of a possible implementation, this can be used as a placeholder as long as the implementation is not available. However, behavioural descriptions are limited to the behavioural constructs the HDL has to offer. This usually limits statements of specifications to high-level programming language constructs, often in an imperative style, whereas more abstract (non-algorithmic) specifications are often wanted. Another shortcoming is that no proof is provided that the implementation has the same behaviour as the behavioural description placeholder. Moreover, a behavioural placeholder and the final implementation can only be proved correct by simulation as both are given in the HDL. This problem arises every time we have a new implementation at another level of abstraction.

Another shortcoming of using fixed circuit descriptions is that often we would like to describe parametrised hardware. Hardware may be parametrised on a particular subcircuit (plug-in component), and regular hardware may often be parametrically defined on the size of the data. Although this is possible in a number of HDLs (*e.g.* ELLA and VHDL), it is not possible to simulate (or prove correct) parametrised descriptions: they must be instantiated to a fixed number of bits, or must use particular plug-in components.

For these reasons we would like to move to a more powerful system, where we can use HDL descriptions for (symbolic) simulation, but may also use general formulae as specifications of abstract components, verify hardware which is independent of a particular word size, *etc.* A formal relation between specification and implementation must be provided so that we can safely replace a specification, used as a placeholder, by a particular implementation. For example, formal proof of correctness of abstraction functions is an important facet of hardware design, documentation, and verification.

The field of formal hardware verification has tried to address some of the problems raised above. In the next section we will briefly give an introduction to formal hardware verification. Following this, we present the approach taken in this thesis which intends to address the problems inherent in simulation, and offer a solution which fuses the formal treatment of hardware verification and widely used hardware description languages.

1.2 Formal Hardware Verification

The formal verification of hardware attracted interest in the early 1980s in response to the concerns discussed in the previous section. Formal notations and automated proof systems have been used in a variety of methodologies. These include formal logics (*e.g.* first-order [98,34], higher-order [84,79], temporal [139,171]), type theory [102], state machines and automata [81,28,15], process algebras (*e.g.* CIRCAL [132], HOP [74]) and others [56,155,165]. Many other formalisms and systems have been studied, some of which are reviewed in [66,35].

The avoidance of the combinatorial explosion of test vectors, mentioned above, is a major driving force. For example, if a logic is used, symbolic variables are available which allow symbolic simulation [73]. A proof system will often have capabilities such as simplification of (intermediate) formulae built-in. Moreover, most logics allow mathematical induction. For example, it becomes possible to prove a component correct for *all time* [80] without having to simulate the design for ever. Similarly induction may be applied to prove correctness of parametrised hardware [83]. Thus an N bit adder may be proved correct once, instead of verifying every instantiation for a particular word size. Another important aspect of hardware design is abstraction [121]. For example, structural abstraction corresponds to circuit decomposition. Data abstraction is a crucial and recurring theme. When a circuit design proceeds to a lower level of abstraction in a top-down design method data values are often encoded in a different manner; natural numbers may become bit vectors, for example. This abstraction must be dealt with in HDL descriptions in order to compare simulation results arising from implementations at different levels of abstraction. Although some HDLs provide facilities to insert abstraction functions at module boundaries there is no formalised correspondence between the levels of data representation. Using a formal logic a description of various abstractions may be given, and proved correct.

While the hardware verification field has been relatively successful in dealing with theoretical issues, industrial take-up of these methods has been limited. Partially this is due to the problems of scaling up models which have been used for hardware description. Industrial sized designs have not been verified yet. (The largest circuit to date has probably been the VIPER chip, which has been partially verified [41,42].) The use of many different notations has also hindered industrial acceptance. A major motivation for this thesis has been the integration of formal approaches advocated by the hardware verification community with commonly used notations used by practising hardware designers. We hope to achieve this through the provision of a standard HDL in a proof system [72].

1.3 Combining HDLs and Formal Verification

In our work we integrate an HDL and a higher-order logic proof system. A number of different paradigms may be combined into one overall framework by doing so. Consider the following diagram.

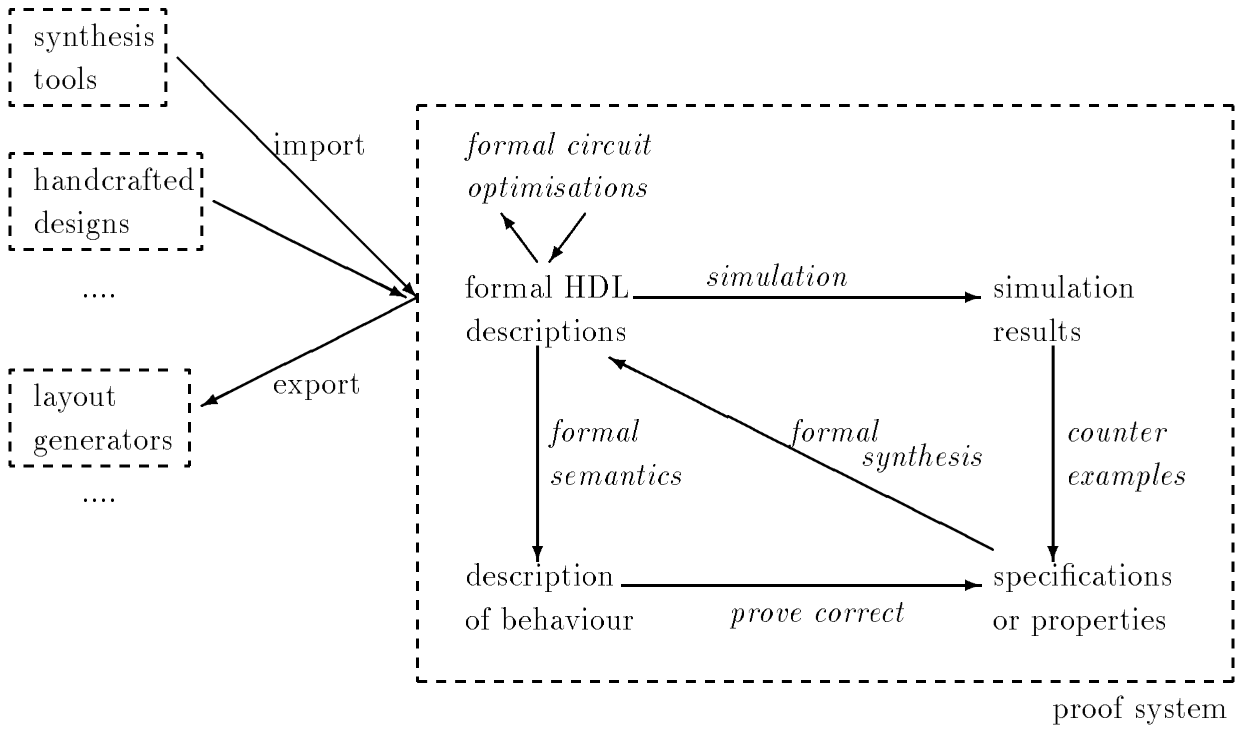

Figure 1–1: Overview of Our Approach

To *embed* an HDL in a proof system [72] we provide objects of type HDL (or circuit.) These are purely structural entities which have no meaning. A separate object, the *formal semantics*, provides the mapping from circuit descriptions to their behaviour. Results may be proved about particular HDL descriptions by mapping them onto their behaviour and then using this to prove correctness properties. Moreover, it is possible to prove properties about the semantics itself. There may be more than one semantics, in which case we can prove facts about the relationships between them. The embedded semantics may also be used as a sophisticated symbolic simulator, incorporating all the properties described in earlier sections. Abstract hardware and formal links between specifications and implementations, and hardware descriptions at different levels of abstraction are all possible.

In addition, transformations on circuits may now be conducted on familiar HDL descriptions, and may be formally proved correct. This can be used in conjunction with transformational design methodologies [30] or *post hoc* circuit optimisations [115].

Various approaches to synthesis also benefit from using an HDL in a proof system. As circuit descriptions are part of the proof system they can be manipulated like any other proof system entity. In particular, functions can operate on and return circuits. Hardware generators or synthesis tools have been formally verified using the Boyer-Moore theorem prover [22], NUPRL [9] and HOL [38].

Interactive methodologies such as Hanna's formal synthesis [86] can be adapted to manipulate HDL descriptions rather than logical formulae. The resulting design is correct by construction, and no translation is necessary to export the synthesised designs into the 'real world.' Fourman *et al.* use a refinement-based approach, where a circuit design is gradually refined and instantiated along with a proof of correctness [62,59]. Although it uses a relational form of hardware description, common in the higher-order logic hardware verification community, this could be adapted to use HDL descriptions instead. Together with the graphical interface used in [62,59] this would allow hardware designers to schematically construct formally verified designs written in a familiar notation.

The use of a widely used HDL facilitates the communication with other tools. Designs may be exported to layout generators, or imported from synthesis tools to be verified.

The objective of the research described in this thesis is to construct a prototype system to test the feasibility of the approach described above. A clean embedding is therefore important. Whenever a conflict between theoretical advantages and practical issues arises we invariably discard usability in favour of theory. For example, the restriction to a small HDL subset introduces difficulties when describing large circuits. However, we might not have been able to even attempt to describe large circuits had we started with a full-blown industrial HDL. After constructing the basic infrastructure, *i.e.* embedding the HDL, we attempt to use the different methodologies, such as formal synthesis and symbolic simulation, to assess the extent to which they have been combined into a unified framework.

1.4 The Structure of this Thesis

In Chapter 2 we describe our perception of the separation of structure and behaviour of circuits. This leads us to a particular type of embedding of an HDL in a proof system. Related work is discussed throughout the chapter.

We choose a particular HDL subset to use in the remainder of the thesis in Chapter 3, after reviewing two candidate HDLs. A formal semantics of the subset is then presented, followed by a discussion of possible alternative semantics and related work.

In Chapter 4 we give an introduction to the proof system we use, and show how the HDL subset can be combined with this proof system. Some important results which have been proved about the semantics in the proof system will be reviewed.

A number of small examples are presented in Chapter 5. It is shown how various methodologies such as symbolic simulation, interactive hardware synthesis, and synthesis functions may be used in conjunction with the embedded semantics for an HDL.

Finally, in Chapter 6 we conclude and present directions for further work.

A number of appendices have been included. Appendix A contains an overview of terminology and notation used in this thesis. Conventions for displaying proof system output are described there. We would like to draw special attention to indices (pages 240–244) of the definitions of types, functions and abbreviations which are used.

Appendix B contains an overview of the lemmas and theorems which have been proved about the embedding in the proof system.

Appendix C lists a number of proof system rules which implement the formal semantics of the HDL subset presented in Chapter 4. Some pretty-printing conventions, alternative semantics rules, and a correspondence between paper and embedded versions of semantics rules are also given there.

Chapter 2

Integrating an HDL and a Proof System

In this chapter we describe the reasons for embedding an HDL in a proof system against a historical and evolutionary background. We describe the relation between the structure and behaviour of a circuit description, and how this relation has been implemented in previous work.

2.1 Relating Circuit Denotations and Behaviour

In our opinion the most important way to make meaningful statements about actual hardware is to talk about its behaviour under a set of input stimuli. Input stimuli produce observable output excitations. The relation between input and output is the *behaviour* of the device. We take the behaviour of systems as fundamental.

Hardware description languages were developed to describe particular designs, which could then be implemented. The first rôle of HDLs was to document designs [100]. Older HDLs only provide facilities for structural descriptions — no means for describing behaviour is provided. Later it was realised that these descriptions could be used to *simulate* the realisations of the designs they described. The shift from the use of HDLs as a documentation tool to their use as behavioural descriptions is important [152]. A structural description of the physical realisation of the system was replaced by the behavioural description of the design of the system. Low-level HDLs, such as EDIF [51] continue to be used primarily to describe the connectivity of circuits. Higher-level HDLs use behavioural descriptions exactly because they express what the design is supposed to do without having to resort to an intermediate structural description which has to be simulated.[1] Thus there is a large gap between an HDL description and the behaviour of one of its implementations. This gap is bridged by its simulation model. The accuracy of the correspondence between a circuit's behaviour and the simulation behaviour of its HDL description depends crucially on the model which is used [43,26]. All models are abstract and hence cannot be used to draw meaningful conclusions concerning features from which they abstract. A description of a design is therefore not just verified; it is *verified with respect to a specification using a particular model*. For example, Joyce *et al.* reveal in [106] that a gate-level model used in the verification of a microprocessor included unrealistic power-up assumptions. Using a more accurate switch-level model an error in the design was discovered. The subsequent correct functioning of the implementation underlines the fact that the *more detailed* model did not accurately describe the implementation's behaviour. Henceforth we assume that HDL descriptions are interpreted relative to some model. We will not concern ourselves with the accuracy of this model.

[1] However, this behaviour can be accessed only indirectly, using a simulator. This is not the case for all behaviour descriptions. Mathematical logic may be considered expressive enough and at a sufficiently high level to serve as a behavioural description which does not need to be animated.

Structure and Behaviour in Formal Hardware Verification

Where proof assistants have been used in the hardware verification community, the following schema has generally been used to describe circuits.

$$\vdash \textit{implementation} \text{ IMPLEMENTS } \textit{specification}$$

The relation IMPLEMENTS expresses that the implementation satisfies the specification. IMPLEMENTS has been interpreted as equivalence ($\leftrightarrow$ or $=$), and logical implication ($\rightarrow$.) Although more sophisticated notions have been investigated [185, 179,13], logical implication is used predominantly. Nearly always *implementation* is a relation, describing the behaviour of the design under consideration. Although this poses no problem in principle, in practice this behavioural description of the implementation has been regarded as a structural description. However, in purely structural descriptions there is no behavioural information. `and(x,y,z)` means only 'the piece of hardware commonly called an AND gate.'

In the approach taken by researchers using the Boyer-Moore theorem prover [21] `and(x,y,z)` already denotes a particular behaviour — that normally associated with an AND gate (at gate level.) Consider the following representative example from [22].[2]

```
(defn b-not a) (if (equal a F) T F)
(defn b-and a b) (if (and (boolp a) (boolp b))
                     (and (equal a T) (equal b T))
                     F)
(defn b-nand a b) (b-not (b-and a b))
```

The behavioural description has been broken down into small components which we relate directly with their usual implementations, but strictly speaking the description is still behavioural. The example above consists of a composition of

[2]Whenever we show output from related work, we will keep as close to the original notation as possible. Unfortunately, this means that something as simple as implication may be written as $\rightarrow$, $\supset$, $\Rightarrow$, or *implies*.

constants which already have an interpretation. In our approach we insist on beginning with the uninterpreted syntax of a structural language. Behaviour is a secondary concept, and is provided in an explicit manner. This highlights the fundamental difference between the structure of hardware and the behaviour of the hardware, when it is abstracted using a particular model. In the Boyer-Moore system only Brock and Hunt have used this approach [22]. Their work will be discussed below, in Section 2.3. Other Boyer-Moore work provides interpretations such as the one given above. The hardware description is a recursive function which is intended to model the behaviour of the design. Hunt first used tail recursion to represent the advance of time [98]. The same idea has been used by later hardware verification research based on the Boyer-Moore theorem prover; for a general account of this method see [146].

In the HOL proof assistant [79], nearly all work has been in terms of similar direct interpretations [83, *e.g.* Section 4]. The exceptions will be discussed later in sections 2.2 and 2.3. Consider the usual HOL definition of an AND gate [44]: $\vdash \texttt{and}(x, y, z) = (z = x \wedge y)$. It defines a three-place relation between booleans. It may be composed with a similarly defined NOT gate as follows:

$$\vdash \texttt{and}(x, y, z) \wedge \texttt{not}(z, a) \tag{2.1}$$

Although this looks conspicuously like a structural description it is a behavioural description, composed of the two very simple relational descriptions `and` and `not`. In the LAMBDA[3] proof assistant [64], used in later chapters, the example would read as follows. The definition of `and` would be `val and#(`x`,`y`,`z`) = (`z` == `x` &&` y`)`. It is combined with a NOT gate as follows:

$$\vdash\ \texttt{and\#}(x, y, z) \wedge\ \texttt{not\#}(z, a)$$

Where `&&` is the boolean conjunction, and $\wedge$ is the conjunction for truth values. Truth values have type Ω rather than type *bool*. (See Section 4.1 for more information about LAMBDA.) A corresponding description which is truly structural

[3]LAMBDA is a product of Abstract Hardware Ltd.

would be given in the following style:

$$\vdash \ \texttt{P\#(strand(x,y,z)} \wedge_{str} \texttt{strnot(z,a))} \tag{2.2}$$

There is a considerable difference between the previous relational behavioural description and this purely structural description. `strand` is an object denoting a purely structural AND gate. $\wedge_{str}$ is an operator combining structural descriptions, with result type *structural*. The purely structural description is not a truth-valued expression, like the relational descriptions. We have to say something about the structural expression, which is what the context `P` indicates. We could give a meaning to the structural description using a behaviour extraction function. Given a circuit, such a function returns a description of its behaviour. Assuming $\texttt{behaviour} : (name \rightarrow bool) \rightarrow structural \rightarrow \Omega$ is a homomorphism (*i.e.* maps $\wedge_{str}$ to `&&`), we would have:

$$\begin{aligned}
&\texttt{behaviour}\ env\ (\texttt{strand}(x,y,z)) \ = \ (env\ z \ = \ (env\ x)\ \texttt{\&\&}\ (env\ y)),\\
&\texttt{behaviour}\ env\ (\texttt{strnot}(x,y)) \ = \ (env\ y \ = \ not\ (env\ x))\\
&\vdash \ \texttt{behaviour}\ env\ (\texttt{strand}(x,y,z) \ \wedge_{str} \ \texttt{strnot}(z,a)) \leftrightarrow\\
&\qquad env\ a \ = \ not\ ((env\ x)\ \texttt{\&\&}\ (env\ y))
\end{aligned} \tag{2.3}$$

not is the boolean NOT operator. $env : name \rightarrow bool$ is an environment or valuation function, giving a value to a name. Later we will see this same pattern arising; the use of a semantic function to give a meaning to structural descriptions, and an environment to link structural names to values.

It may be very useful to have multiple behaviour functions, emphasising different aspects of the structural description [22,140,162]. For example, a circuit layout could be extracted if the $\wedge_{str}$ operator included placement and routing of components.

In our opinion a proper separation between the structural and behavioural aspects of a circuit description is crucial. In the remainder of this section we review research which has explicitly addressed this issue.

2.1.1 Research Relating Structure and Behaviour

In [84] Hanna and Daeche present the VERITAS hardware verification approach which includes a proof assistant implementing a classical polymorphic higher-order logic. *Theories* are used to define new notions, starting with propositional logic, followed by theories defining natural numbers, time, waveforms, *etc.* Waveforms are analogue and may or may not correspond to a discrete digital value at particular points in time. A theory of *gate behaviours* contains definitions for basic gates. It is important to note that only behaviours are defined, there is no mention of any particular implementation or structure. For example, if *wf* is the type of waveforms,

$$\vdash \textsc{andbehav} : \ \mathit{characteristics} \rightarrow (\mathit{wf} \times \mathit{wf} \times \mathit{wf} \rightarrow \mathit{bool}) \ = \ \mathit{definition}$$

is a parametrised relational definition of the behaviour of an AND gate. An example of *characteristics* could be the timing characteristics of the particular AND gate behaviour. The three waveforms correspond to two input and one output waveforms. However, note that strictly speaking this is a structural interpretation which we cannot express yet. The association of structure with behaviour can only be completed after a theory of *simple structures* has been defined. This theory defines the purely structural aspects of a circuit. Input and output ports, components and interconnects are axiomatised through the notion of subtypes. The most general structural type *struct* has a subtype *port*, which in turn contains subtypes *inport* and *outport*. The *gate* subtype of *struct* contains types of components such as the type *andgate* of AND gates. For each type there may be a number of implementations; *i.e.* elements of the type. To reason about these purely structural entities projection functions are used. Projection functions extract particular characteristics from a value. For example, we use the function $in_i : andgate \rightarrow inport$ to obtain the ith input port of an AND gate. A waveform function $W : port \rightarrow wf$ may then be used to access the waveform of a port. Thus, basic components may only be used through associated projection functions which extract a particular property. We associate an AND gate behaviour ANDBEHAV, as defined in the gate behaviour theory, with a particular simple structure g of type *andgate* as follows.

$$\vdash \forall g : andgate.\ \text{ANDBEHAV}\ (characteristics\ g)\ (in_1\ g)\ (in_2\ g)\ (out\ g)$$

This axiom states that every purely structural AND gate g with its particular properties, in this case *characteristics* g, input and output ports, satisfies the behaviour of an AND gate as axiomatised by ANDBEHAV. Finally, a theory of *compound structures* defines hierarchically composed structures. Properties of structures, such as subgates and their interconnections, are again obtained by applying projection functions. However, by virtue of being composed of more basic components, the behaviour of composite structures may be derived by using the behaviours of their subcomponents. This work is a good example of the separation of structure and behaviour. It is very distinctive in its use of projection functions to extract the composition of compound structures. The more usual approach is to define compound circuits by the explicit composition of substructures, using composition and hiding operators (*e.g.* CIRCAL [132] and LCF_LSM [81].)

Wang [177] describes a Hardware Synthesis Logic, which also maintains a clear distinction between structure and behaviour. Circuit structures are composed in a simple structural algebra, called the implementation language, containing a structural connective &, which is comparable to $\wedge_{str}$ introduced on page 14. A logic, called the specification language, is used to reason about properties of implementations and about specifications. The calculus is independent of a particular specification logic, although a higher-order logic is used in the example below. The implementation and specification languages are related through a so-called construction logic, which comprises two basic axioms, a small number of inference rules and some axiom schemas. Axiom schemes define, using the specification language, the behaviour S of basic terms I in the implementation language. This is denoted by the use of the connective in $I \models\!\mid S$. It may be used to state the behaviour of a register as follows:

$$Register(i, c, o) \models\!\!| \ \forall t.\ o(S\ t) = \mathit{if}\ c\ t\ \mathit{then}\ i\ t\ \mathit{else}\ o\ t$$

In this work the structural conjunction & is preserved by $\models\!\!|$, so that we have the following inference rule in the calculus:

$$\frac{\begin{array}{c}\vdash I_1 \models\!\!| \ S_1 \\ \vdash I_2 \models\!\!| \ S_2\end{array}}{\vdash I_1 \& I_2 \models\!\!| \ S_1 \wedge S_2}$$

It seems reasonable to assume that the behaviour of a component is composed of the behaviours of its subcomponents. This corresponds to the assumption for Equation 2.3 of page 14 that a behaviour function is a homomorphism. Wang proves a number of meta-results relating the implementation, specification, and construction logics.

The formal synthesis methodology of Hanna *et al.* [86] has similar features to Wang's work, but takes place within the VERITAS logic. As a result, the distinction between structural and behaviour composition is lost: both are represented by the same logic conjunction operator (*e.g.* the **split** rule in [86].) This work is distinct from earlier research [84] described above.

Melham discusses the separation of structure and behaviour in Sections 3.1 and 3.2 of his thesis [124]. In most of [124] the standard HOL behavioural descriptions which we showed in Equation 2.1 are used. However, a means to denote the structure of circuit separately from its behaviour is introduced to discuss the notion of abstraction between models of hardware behaviour. We will review this aspect of Melham's work in the next section.

2.1.2 The Programming Language Semantics Connection

The structure versus behaviour issues discussed above have been investigated thoroughly for conventional programming languages in the formal semantics field. Circuits are called programs, and there is no concept of time. Three types of semantics have been proposed to give meaning to programs; axiomatic [60,95], denotational [169,159,173] and operational [150,108].

Axiomatic semantics gives properties of a program directly, without the need to evaluate the program. Two examples of very direct applications of a Floyd-Hoare style axiomatic semantics to HDLs are given in [149] and [120]. We review this work in Section 2.2.1. The various kinds of behaviour functions given above are also examples of the axiomatic approach. An advantage of this approach is that no evaluation model is needed to arrive at properties of structural terms. This means that the intended behaviour is independent of any simulator model one may wish to use.

A denotational semantics is intended to map a program directly onto its meaning or denotation, which is generally taken to be a function from stores to stores. Input-output relations can be derived by reasoning about the functions. Circuits with the same meaning are mapped onto identical objects.

The term 'operational semantics' indicates a family of semantics which includes Plotkin's Structural Operational Semantics [150] and Kahn's Natural Semantics [108]. We will use the term operational semantics to indicate the relational style of semantics, rather than only Structural Operational Semantics. In operational semantics different implementations with the same meaning (*e.g.* input-output relation) may be distinguishable by their internal behaviour. This is a result of taking a very computational approach. The meaning of a program in this setting is given by the (labels of the) transitions the program performs during its evaluation. More detailed results can be proved about programs using an operational semantics, because it takes the simulation or evaluation algorithm into account.

The three types of semantics are suited to different applications [159]. Axiomatic semantics map programs directly onto properties characterising their behaviours. Denotational semantics map programs onto functions, from which input-output behaviours may be derived. Operational semantics allow a behaviour to be derived through the sequence of transitions a program may perform.

The remainder of this chapter is structured as follows. The next section reviews direct mapping of circuits to behaviours corresponding to an axiomatic semantics approach. In Section 2.3 we adapt the operational semantics approach to HDLs. The behaviour of HDL descriptions may be then extracted indirectly. Related denotational semantics work will also be discussed there.

2.2 Extracting Behaviour From Circuit Descriptions

The intuitive solution to the structure-behaviour division described above is to directly *extract a behaviour* from a circuit description. We have a function `behaviour` : *structural* $\rightarrow$ Ω. Here Ω stands for the type of truth values. In other words, `behaviour` maps a hardware description to a logical formula characterising its behaviour. In our examples, we will generally use some higher-order logic to map HDL descriptions into. For example,

$$\texttt{behaviour}\,(\texttt{delay}(\texttt{c}, \texttt{in}, \texttt{out})) \;=\; (\mathit{out}\ 0 == \texttt{c}\ \wedge\ \forall t.\ \mathit{out}(\texttt{S}\ t) == \mathit{in}\ t) \quad (2.4)$$

Here `delay(c,in,out)` is an HDL description for a unit transport delay. Let us first assume that this equation is entirely outside a proof system. This definition raises the following question: what is the relation between `in` and *in*? The former is a syntactic structural object, whereas the latter is part of the formal system in which the behaviour is expressed. The situation is clarified by giving explicit types to the various components:

$$\begin{aligned}
\texttt{behaviour} &: \mathit{structural} \rightarrow \Omega \\
\texttt{delay} &: (\mathit{value} \times \mathit{name} \times \mathit{name}) \rightarrow \mathit{structural} \\
\texttt{in} &: \mathit{name} \\
\mathit{in} &: \mathit{signal} = \mathit{time} \rightarrow \mathit{value}
\end{aligned}$$

Although this behaviour function resembles that of Equation 2.3 of page 14 it is very different (even ignoring, for the moment, that the latter is part of a proof system.) In Equation 2.3, x, y and z appear in both the domain and range of the behaviour function. In the range, however, we use the value associated with the name via the environment. In Equation 2.4 the behaviour function does not establish any explicit relationship between `in` and in. We would argue that the behaviour function of Equation 2.4 provides a higher-level axiomatic semantics than the behaviour function in Equation 2.3. The other function is closer in spirit to a denotational semantics.

The definition of `behaviour` could be an entirely informal exercise. A number of successful implementations of this approach are described below. Later work advocated the use of a proof system to map the extracted behaviour into instead of an *ad hoc* implementation of the manipulation of behaviours. This led to a clean separation of conceptually different processes, namely the extraction of the behaviour and the formal reasoning about this behaviour. We may view Equation 2.4 in this light; the right hand side would be inside the proof system. Equation 2.3 cannot be interpreted in this manner, however. This is due to the appearance of x, y, and z in both the domain and range of `behaviour`. One fundamental problem remains: the behaviour function itself resides outside the proof system. This means that we cannot reason about it within the proof system. In particular, we will have to accept the correctness of the implementation of the behaviour function in good faith. The HDL description is also informal, which means we cannot reason about structural terms. We can only use the behaviour of the design, and not its structural description, inside the proof system. This becomes a problem where we want to reason about general properties possessed by all, or a set of, circuits. The solution is to move the behaviour function into the proof system also. For example, some hardware models may satisfy the property that for every input a (possibly unique) output exists. Rather than proving this for every instance we are interested in, we would rather prove a theorem of the form

$$\vdash \forall e : structural.\ \forall i : const.\ \exists o : const.\ simulation\ e\ i = o$$

It is important to note that the type *structural*, representing terms of type circuit, resides inside the proof system. We may then quantify over these terms and state properties involving all possible circuits using $\forall$. We have a real separation between the description of the circuit and its behaviour. The circuit description is purely structural, and it may be manipulated independently from its behaviour. We will discuss this aspect of embedded circuit descriptions in more detail in Section 2.3. It is not always possible to move the behaviour function into a proof system. In the Boyer-Moore proof system there is no problem because it is not strongly typed. Everything is encoded using lists, which may be interpreted in various ways. Truth values and judgements are not distinguished from other objects. In the HOL system this is also possible because functions can produce and manipulate truth values. In the LAMBDA proof assistant it is not possible for functions to operate on truth values (see Section 4.1.1); as a result no useful behaviour function can be encoded.

Consistency, soundness, and completeness are properties which are important for behaviour functions. These are meta-theoretic properties in the sense that we cannot prove them within the formal system we use. We would like behaviour functions to always produce formulae which are consistent, *i.e.* do not contain contradictions. If this were not the case, a particular circuit for which an inconsistent behaviour description was produced would satisfy any specification. This is the 'false implies everything' problem [185,179,13].

Soundness means that we cannot derive any false results using the behaviour function. If our intuition is that c is an adder, we would like to prove, using the behaviour function output, that it does indeed add two numbers. This is a semantic question in the sense that it is our interpretation which provides the ultimate truth. Unless we have another semantics with which we can compare the behaviour function, soundness is a meaningless concept. As claimed previously, we could take one of the semantics as the definitive semantics, and check the others against it.

The same applies to completeness. Completeness states that we do not lose any information about the behaviour of the circuit when we apply the behaviour function to it. As an extreme, we could map every circuit to a trivial input-output relation, but this would not help us in deriving useful results about circuits. We note that in principle the range of the behaviour function may be anything, as long as it allows us to express the intuitions we have about the behaviour of circuits. Higher-order logic is a good candidate for hardware description, because many concepts, such as time, signals, and hidden wires, may be conveniently expressed in it [83].

2.2.1 Behaviour Extraction in the Literature

The remainder of this section will review research which has some aspect of explicitly relating a circuit description to its behaviour.

Floyd-Hoare Style Axiomatic Semantics

Floyd [60] and Hoare [95] introduced axiomatic semantics, in which programming language constructs are given a meaning through the use of pre- and post-conditions. For example, a while loop has the following semantics:

$$\frac{\{P \wedge Q\}\ c\ \{P\}}{\{P\}\text{WHILE } Q \text{ DO } c \text{ OD } \{P \wedge \neg Q\}}$$

This rule states that if P is the loop invariant, then we can infer that after the while loop terminates the pre-condition P is still true and the loop condition Q is false. Note that the proof that the loop terminates must be given separately. A number of conventional programming languages have been given a formal semantics in this style [1,163]. Pitchumani and Stabler [149] used a Floyd-Hoare style semantics to give a definition for a register-transfer level HDL. The main difference between conventional programming languages and hardware description languages is the notion of time in the latter. The language which is described in [149] does not have an explicit notion of time: there is no delay construct for example. Rather, time is introduced in the semantics through the use of a distinguished variable t, representing time. t may be used in pre- and post-conditions, but not in programs. This precludes assignments to the time variable, but does allow temporal information to be given in the specification, for example. Consider the NULL statement. Its conventional semantics is $\{P\}$ NULL $\{P\}$. When time is involved this becomes $\{P[t+1/t]\}$ NULL $\{P\}$. NULL has no effect other than to pass time. In other words, everything that was true before executing the NULL statement at time t, is also true after it has finished, at time $t+1$. A drawback of this approach is that users have to instantiate the rules with particular formulae P, Q themselves.

McFarland and Parker [120] give a Floyd-Hoare style semantics for an HDL called ISPB. ISPB is a behavioural language, used to describe processors at the instruction level. The meaning of a ISPB program is the set of interactions between it and its environment. This operational view is formalised using the notions of events, histories, and behaviours. An event is either a read or write statement in the program, a history is a sequence of events, and the behaviour of a program is the set of histories which an execution of the program may produce. Equivalence between processor descriptions is defined as equality of their behaviours. Now *behaviour expressions* are introduced to describe histories in an abstract manner, using regular expressions and data dependencies. The meaning of a behaviour expression is defined to be the set of history $\times$ event pairs satisfying it. Finally, using the Floyd-Hoare triples defining the meaning of ISPB descriptions, a behaviour function is defined. This function maps programs into behaviour expressions: $behaviour : program \rightarrow behaviour\ expressions$. This procedure has been shown to be complete. In other words, the behaviour expression produced by the behaviour function defines a set of histories; this set turns out to be equal to the set of all histories obtained by executing the program using the operational definition. Behaviour expressions are then used to prove that transformations on programs are correct, *i.e.* behaviour preserving.

Ad Hoc Behaviour Extraction Functions

Early research into hardware verification was informal and rather *ad hoc*. Most efforts took the form of a software system which, given a hardware description and a specification, would try to show their equivalence. Specifications were either at the same level of abstraction as the hardware description, or one level above. Hardware was usually described at the transistor level, gate level or register-transfer level (RTL.) Specifications were mostly given in terms of boolean functions. From a historical perspective we may consider these efforts as primitive behaviour extraction functions.

In the late 1970s and early 1980s a number of efforts were directed at *functional abstraction*. This is the name given to the process of extracting a behaviour from a circuit description. In [112] Leinwand and Lamban describe an automated system which extracts a dynamic boolean algebra description from an implementation at the asynchronous-transistor level. Dynamic boolean algebra consists of classical two-valued boolean algebra augmented with rise and fall operators. A second phase matched subexpressions with dynamic boolean algebra descriptions of standard circuitry, such as adders. The resulting asynchronous sequential behaviour was then rewritten to a synchronous behaviour with as few states as possible. Using the notion of *weak simulation* [128] the extracted behaviour and specification could be compared at the boolean-function level. It is a remarkable achievement that this temporal abstraction [121] was automatically dealt with.

Weise [178] describes the Silica Pithecus system, which extracts functional three-valued boolean behaviour from MOS circuit descriptions. In addition to the circuit's description and its specification, a set of constraints is part of the system's input. There are three types of constraints: (i) Those asserting the validity and interrelationships of inputs. (ii) Constraints ensuring valid outputs; these are derived by the system using the data abstraction function mapping analogue signals to booleans. (iii) Constraints which state the meaning of basic MOS components. The latter axiomatise the behaviour of the basic building blocks. Rather than manipulating circuit models at the lower signal level, only the abstracted behaviours are used. Moreover, instead of using constraints describing the behaviour of a circuit, constraints which define the boundaries of correct behaviour are used. The system verifies subcircuits only once; multiple occurrences use the same constraints. The hierarchical verification and use of abstracted instead of detailed behaviour make this system efficient.

Other related work which did not use a proof system includes [171,7,186,117].

Partially Formal Behaviour Extraction Functions

In [14] Borrione and Paillet recognise the need for a formal system to unambiguously express the semantics of an HDL. They outline the design of a system to translate VHDL descriptions to a representation of their behaviour in a proof system. The behaviour is represented by a set of simultaneous functional equations, in the Boyer-Moore and REVE proof systems [146]. As indicated earlier in this section, the use of a proof system is a great improvement on the *ad hoc* implementations reviewed so far.

Boulton *et al.* [19] describe a behaviour extraction function from a subset of ELLA to the HOL proof assistant. The behaviour function and its abstract syntax tree input are outside the formal part of the HOL proof system. The behaviour function has been kept simple in order to minimise the risk of errors. This is possible by mapping ELLA constructs to high-level behaviours in HOL. For example, consider the multiplexor, called a `case` statement in ELLA.

$[\![$`case in of lo:  hi, hi:  lo`$]\!]$ =

`CASE` $[\![in]\!]$ `[OF [`$[\![$`lo:  hi`$]\!]$`;` $[\![$`hi:  lo`$]\!]$`]] (UNLIFT UU) =`

`CASE` in `[OF [CONST lo, SIGNAL LIFT_hi;`

`             CONST hi, SIGNAL LIFT_lo]] (UNLIFT UU)`

$[x, y; a, b]$ is a list containing two pairs (x, y) and (a, b). $[\![\cdot]\!]$ is the behaviour function giving a semantics to the structural description (`case in of lo:  hi, hi: lo`.) `CASE` and `OF` are HOL functions which together represent the behaviour of the whole `case` statement, given the subcomponents' behaviours ⟦`lo: hi`⟧ and ⟦`hi: lo`⟧. Because this behaviour function is itself not part of HOL, the `case` statement is informal, and the variable `in` has no explicit relation to *in* (*cf.* Equation 2.4 on page 19.) [19,17] go into some detail describing the semantics of ELLA, which we will comment on in Section 4.2.4.

Other related work includes [174,56] which describe mapping VHDL into HOL and SDVS respectively. SILAGE has also been given a HOL semantics in this manner [75,2]. In [145] the Boyer-Moore and TACHE theorem provers are used as the target logics to map behaviours of CASCADE descriptions into. Eveking uses the LOVERT system to check the equivalence of SMAX HDL circuit descriptions [54,55]. Recently Umbreit has mapped VHDL programs onto ML descriptions using the LAMBDA proof assistant [172].

Formal Behaviour Extraction Functions

In the work described above the circuit descriptions and behaviour function were not part of a proof system. Here we describe research in which the behaviour function is formalised also.

In [123,124] Melham describes a formal behaviour function in HOL. He defined an abstract data type representation of CMOS circuit descriptions inside the HOL proof assistant, using a recursive data type definition package [122]. Part of this data type is given below.

$$circ ::= \texttt{pwr}\ str \mid \texttt{ntran}\ str\ str\ str \mid \texttt{join}\ circ\ circ \mid \ldots \qquad (2.5)$$

The intuition for these structural combinators is as follows. `pwr` w indicates that the wire name w is connected to power. `ntran` $g\ s\ d$ describes an N-type transistor with wire names g, s, and d as its gate, source, and drain respectively. `join` $c\ c'$ is structural composition, comparable to $\wedge_{str}$ of page 14. A switch-level model and threshold model semantics were given to these structural descriptions using primitive recursive functions. These semantics model circuits at one point in time only. We will give only a flavour of the definitions for the switch-level semantics.

$$
\begin{aligned}
&\vdash \texttt{Sm}\,(\texttt{pwr}\ p)\ e && = (e\ p = T)\\
&\vdash \texttt{Sm}\,(\texttt{ntran}\ g\ s\ d)\ e && = (e\ g \supset\ (e\ d = e\ s))\\
&\vdash \texttt{Sm}\,(\texttt{join}\ c_1\ c_2)\ e && = \texttt{Sm}\ c_1\ e \wedge \texttt{Sm}\ c_2\ e
\end{aligned}
\tag{2.6}
$$

T is the boolean value true, and $\texttt{Sm} : \mathit{circ} \to (\mathit{str} \to \mathit{bool}) \to \mathit{bool}$ is the function mapping circuits together with an environment to a formula describing their switch-level behaviour. The term $e : \mathit{str} \to \mathit{bool}$ is the environment, or valuation function, mapping strings *str*, denoting wire names, to their values. As we briefly indicated on page 20, because the data type expressions are ordinary proof system terms we may quantify over structural descriptions. This feature was used to relate the switch-level and threshold models of hardware formally, *i.e.* as a theorem in HOL.

In [8] Basin uses the NUPRL proof assistant [46], which implements a constructive type theory. He uses the *proofs-as-circuits* paradigm, which is an adaptation of the *propositions-as-types* idea [97]. A constructive proof contains computable evidence, *e.g.* a program, of the truth of the proposition it proves. Different proofs of a specification correspond to different implementations. For example, proving

$$>>\ \ \forall i, o.\ \exists c.\ S(i, o, c)$$

entails exhibiting a witness c which satisfies the specification $S(i, o, c)$. (>> is NUPRL's judgement.) However, there is no guarantee that this realisation c has a particular form, or *intention*; we only know that it has behaviour, or *extension*, S. We would like c to be a circuit description, not just any old proof term. To force the realisation to have a particular form, or to be at a particular level of abstraction, a type of circuit terms is introduced. This type *trans* is a recursively defined data type. An interpreter $\mathit{Interp}_{\mathit{trans}} : \mathit{trans} \to \mathit{env} \to \mathit{bool}$ is defined to give a meaning to these terms. *trans* and $\mathit{Interp}_{\mathit{trans}}$ correspond to Melham's *circ* (Equation 2.5) and Sm (Equation 2.6) respectively. A more sophisticated correctness statement may then be defined as follows:

$$>> \ \exists c : trans.\ \forall i, o : bool.\ Interp_{trans}\ c\ env\ \Rightarrow S(i, o)$$

Where *env* is the environment linking the bound variables i and o with their syntactic representations in c (*cf.* e in Equation 2.6, *str* in Equation 2.5.) Note that the circuit description c does not appear in S because $Interp_{trans}\ c\ env$ provides the relation between c, i, and o. In fact, [8] gives a more general correctness statement; parametrised circuits and an arbitrary number of input and output ports are also allowed. To emphasise the use of the behaviour function $Interp_{trans}$, we have also ignored the distinction between the type *bool* and the type of truth values in NUPRL. The definition and use of boolean logic in NUPRL is described in detail in [10].

In the VERITAS approach, discussed in Section 2.1.1, all structural objects are indivisible entities. The behaviour of composite structural objects is not derived through a semantic function defined on the structure of terms as above but projection functions are used instead. The behaviours of subcomponents, obtained through projection functions, are combined to form the derived behaviour of the composite object. Thus in VERITAS only atomic structural objects exist, which may or may not be related to other, perhaps smaller, structural objects through projection functions.

2.3 Deriving Behaviour via a Semantics

In the previous section we showed how behaviour could be extracted directly from circuit descriptions. This is a high-level approach with no indication of an underlying model of how the behaviour is arrived at. Industrial hardware description languages usually have a simulator to animate hardware descriptions written in the HDL. It makes sense not to state properties directly about circuit descriptions, but derive properties using the simulator. That is, we take a more operational stance. Taken at face value, this would seem to imply that we can only derive properties using simulation; exactly what we are trying to get away from. This is not the case, however. If we provide an operational semantics for the HDL, we may prove *general properties* about the simulator model. For example, we would like to prove that the simulator returns an answer for every circuit and set of input stimuli. If this is not the case, a characterisation can be given for circuits or inputs which do not have this property, so that we may avoid them. The operational semantics gives us a firm mathematical grip on the simulator model.[4] Using this approach we can derive behaviour indirectly, via a more intuitive simulation-based route. By proving desirable properties we expect to hold, we can gain confidence in the correctness of the simulation algorithm. These could include totality, monotonicity, some upper bound on the size of the computation, *etc.* We can prove more detailed properties using an operational semantics than using other types of semantics such as axiomatic and denotational semantics. The reason for this is that we can refer to the simulation method, which is absent in axiomatic and denotational semantics. If we also have an axiomatic semantics for the HDL, it becomes possible to prove the soundness and completeness of the operational semantics with respect to the axiomatic semantics. This is achieved by abstracting away from simulation details. As with behaviour functions earlier, we can define an operational semantics on paper, or use a proof system. In this case, however, there is no half-way stage: either everything is on paper, or everything is in a proof system. The reason for this is clear when we consider a fragment of an operational semantics.

$$
\begin{aligned}
&\texttt{opsem}\ env\ (\texttt{wire}\ n) = env\ n\\
&\texttt{opsem}\ env\ (\texttt{parcomp}\ (c_1,\ c_2)) = (\texttt{opsem}\ env\ c_1,\ \texttt{opsem}\ env\ c_2)\\
&\texttt{opsem}\ env\ (\texttt{mux}\ (c_1,\ c_2,\ c_3)) = \textit{if}\ \texttt{opsem}\ env\ c_1\ \textit{then}\ \texttt{opsem}\ env\ c_2\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{else}\ \texttt{opsem}\ env\ c_3
\end{aligned}
$$

`wire` n returns the value on the wire n; note that we assume that $value = bool$.

[4] Particular implementations of this algorithm may still be incorrect. In Chapter 5 we shall show how our approach allows formal simulation, overcoming this problem.

`parcomp` is parallel composition, and `mux` a multiplexor. We see that the variables n and c_i occur in both the pattern and the right hand side. Although it is conceivable to map from outside a proof system into one, this does not really make sense, precisely because the same objects and types occur in both the domain and range of the semantics. This was also the case in Equation 2.3 on page 14, which defined a behaviour function in a denotational manner. This was not the case for the axiomatic behaviour function of Equation 2.4 on page 19.

The discussion which follows applies equally to an embedded operational semantics, and to an operational semantics outside a proof system. The difference between the two is of a more pragmatic nature. It is possible to use an operational semantics on paper to evaluate expressions. However, the size of expressions, environments, *etc.* becomes unwieldy very quickly (see, for example, Section 5.1.) The process is error prone without automation. The use of a proof system relieves the user of the semantics from such mundane tasks as keeping track of environments and variable restrictions. This allows the designer to concentrate on more important aspects, such as the actual design, the proof plan, *etc.* Using the operational semantics should be as painless as possible, *e.g.* by providing commands to apply semantic rules automatically. In the remainder of this section we will take it that our operational semantics is embedded in a proof system.

To embed an operational semantics in a proof system we need to represent circuits, input and output values, and the semantic rules. Auxiliary objects such as environments, and wire names will also be needed. In the previous section, dealing with behaviour functions, we already encountered these notions. However, we reiterate the basic principles here for clarity, and also because these concepts are crucial to our work. The structure and behaviour of hardware are kept separate by providing a structural description language, which is given a meaning through the use of an operational semantics. The operational semantics relates a circuit and its inputs to an output according to some simulation model. The semantics may be nondeterministic or partial. For simplicity we allow only one input. A type for the operational semantics could be the following:

$$\texttt{opsem} : (\mathit{structural} \times \mathit{value}) \times \mathit{value}$$

However, the concept of state is missing; most circuits contain latches, which retain a value between clock cycles. Adding an explicit state yields the following.

$$\texttt{opsem} : (\mathit{structural} \times \mathit{state} \times \mathit{value}) \times (\mathit{value} \times \mathit{state})$$

An alternative view is to dispense with the state, and evolve the circuit itself. The state is part of the circuit description. This is the type of the semantics which we will use in later chapters.

$$\texttt{opsem} : (\mathit{structural} \times \mathit{value}) \times (\mathit{value} \times \mathit{structural})$$

Milner first described this type in [129]. Gordon elaborated it in [78], and used it as the basis for LCF_LSM [81]: the Cambridge LCF extended with a logic for sequential machines to describe hardware. State transition functions of state machines may be seen to have a similar type $(\mathit{value} \times \mathit{state}) \rightarrow (\mathit{value} \times \mathit{state})$. This view is also common in process algebras such as CCS [131], CIRCAL [132], and HOP [74], which use a labelled transition system.

We can state important properties such as soundness and completeness with respect to another semantics. We would like this reference semantics to be more abstract than the operational semantics. If the reference semantics is not embedded in the proof system soundness and completeness cannot be stated within the proof system. In the remainder of this section we assume that there is an axiomatic semantics $\texttt{axsem} : \mathit{structural} \rightarrow \mathcal{P}(\mathit{value} \times \mathit{value})$ available in the proof system as the reference semantics. ($\mathcal{P}S$ is the powerset of set S.) Given a circuit, `axsem` returns a relation of input-output pairs which the circuit defines. The soundness of `opsem` with respect to another semantics `axsem` states that for all circuits, if (i, o) is a valid input-output pair for the operational semantics, it is also in the relation specified by `axsem`. In other words, the low-level `opsem` does not define a larger input-output relation than the high-level `axsem`.

$$\vdash \forall e, e' : \mathit{structural}.\ \forall i, o : \mathit{value}.\ \texttt{opsem}\ (e, i, o, e') \rightarrow (i, o) \in \texttt{axsem}\ e$$

This statement is entirely within the mechanised logic. Completeness states that we do not lose any information: we can derive the same input-output relation using the operational semantics as with the axiomatic semantics.

$$\vdash \forall e : \mathit{structural}.\ \forall i, o : \mathit{value}.\ (i, o) \in \texttt{axsem}\ e\ \rightarrow \exists e' : \mathit{structural}.\ \texttt{opsem}\ (e, i, o, e')$$

Although both semantics define the same relation, we may be able to derive more detailed results using the operational semantics than with an axiomatic semantics. The reason for this is that the axiomatic semantics only defines the input-output relation, whereas the operational semantics defines a mechanism for deriving outputs from inputs. We may prove results about this mechanism.

It is important to realise that the two previous equations manipulate both circuits e and semantics `opsem` and `axsem` inside the proof system. We are able to quantify over circuits, expressing properties which hold for all or particular classes of circuits. For example, it is likely that not all circuits will be well-formed according to some static semantics, and we would like to exclude these circuits from any properties `P` we wish to prove.

$$\vdash\ \ \forall e : \mathit{structural}.\ (\exists t : \mathit{type}.\ \texttt{typeof}\ e = t) \rightarrow \texttt{P\#}(e)$$

Universal quantification may be used, for example, to represent alternative implementations of a circuit. It is also possible to existentially quantify over circuits.

$$\vdash\ \ \exists \mathit{impl} : \mathit{structural}.\ (\exists t : \mathit{type}.\ \texttt{typeof}\ \mathit{impl} = t) \rightarrow \texttt{SPEC\#}(\mathit{impl})$$

By proving this theorem we would define a witness *impl*, which is a well-formed circuit satisfying the specification `SPEC`. In general, all proof system operations are applicable to circuits. For example, circuits may contain free variables, corresponding to plug-in components.

$$\vdash\ \ \forall \mathit{subcomponent}.\ \texttt{PSPEC\#}(\mathit{subcomponent}) \rightarrow \texttt{SPEC\#}(\mathit{plugin}\ \ \mathit{subcomponent})$$

In particular, functions may operate on and deliver circuit expressions; *e.g. plugin* above. This becomes a powerful tool in the form of hardware synthesis functions (Section 5.2.) We can prove the correctness of these functions so that all generated functions satisfy a particular specification:

$$\vdash \forall parameter.\ \mathtt{SPEC\#}(parameter, synthesise\ \ parameter)$$

For example, *synthesise* could be an adder synthesis function. In a similar manner we can prove properties about the semantics, such as totality (Section 4.3.)

We will encounter these features in practice later in this thesis in Chapter 5.

2.3.1 Related Work

Recently a number of hardware description languages have been given formal semantics. With a few exceptions, these have all been paper exercises, and will be reviewed in the next chapter on HDL semantics. In this section we discuss only those which have been used in conjunction with proof systems.

Compiler Correctness Proofs in Proof Systems

Proofs of compiler correctness in proof systems use the same techniques as those used for embedding HDLs in proof systems. To reason about programs their syntax and semantics have to be encoded in the proof system. We will describe Cohn's work [40] in some detail below, because it shows how techniques used by early compiler correctness research have been carried over to later research, reviewed later this section.

Milner and Weyhrauch used the Stanford LCF proof checker to check the correctness of a simple compiler algorithm [127]. The source and target languages were axiomatised in the system through the use of constructors and destructors. Aiello *et al.* encoded a denotational semantics for Pascal in the Stanford LCF in a similar manner [1].

The Stanford LCF was extended to an interactive proof assistant resulting in the Edinburgh LCF [76]. Using the Edinburgh LCF Cohn proved a compiler correct with respect to the denotational semantics of imperative source and target languages [40]. The high-level language was defined using the following types:

$$\begin{aligned} hprogram &= assign + if + while + compound \\ assign &= id \times exp \\ if &= exp \times hprogram \times hprogram \\ while &= exp \times hprogram \\ compound &= hprogram \times hprogram \end{aligned}$$

id is the type of identitiers, and *exp* the type of expressions. These types were introduced by axioms which also defined constructors and destructors such as *mkif* and *destif*. Programs are axiomatised using constructors and destructors, rather than using data types. Research discussed below takes the latter approach. The high-level denotational semantics was defined as the least fixed point of the function *hsemfun*, defined below:

$$hsem : hprogram \rightarrow (store \rightarrow store) = FIX\ \ hsemfun$$

The meaning of a program is a store-to-store mapping. $hsemfun : (hprogram \rightarrow store \rightarrow store) \rightarrow hprogram \rightarrow store$ is defined on the structure of *hprogram* terms. The syntax *if hp = C(v,...)* is syntactic sugar for *if isC hp*, where occurrences of v are replaced by the appropriate projection from *hp*. *eval* is the expression evaluation function.

$$
\begin{array}{lll}
hsemfun = \lambda\ hsem.\ \lambda\ hp.\ \lambda\ s. & & \\
& \textit{if } hp = assign\ (x,\ ex) \textit{ then} & \\
& & s\ [eval\ (ex,s)\ /\ x] \\
& \textit{else if } hp = if\ (ex,\ hp1,\ hp2) \textit{ then} & \\
& & \textit{if } eval\ (ex,\ s) \\
& & \textit{then} \quad hsem\ hp1\ s \\
& & \textit{else} \quad hsem\ hp2\ s \\
& \textit{else if } hp = while\ (ex,\ hp1) \textit{ then} & \\
& & \textit{if } eval\ (ex,\ s) \\
& & \textit{then} \quad (hsem\ hp)\ (hsem\ hp1\ s) \\
& & \textit{else} \quad s \\
& \textit{else if } hp = compound\ (hp1,\ hp2) \textit{ then} & \\
& & (hsem\ hp2)\ (hsem\ hp1\ s)
\end{array}
$$

The LCF is a logic based on continuous functions, and contains a fixed point operator *FIX* with an associated fixed point induction principle. The latter may be used to derive structural induction principles for recursive data types such as *hprogram*. The proofs of correctness proceeded by fixed point induction on the high and low-level semantic functions respectively, followed by a structural induction on program p [29]. Fixed point induction is also called computational induction, because it follows the flow of computation rather than the structure of the program [118]. When we discuss proofs about our embedded HDL semantics in Section 4.3.2 we will mention the similarities and differences with Cohn's proofs.

Other research involving compiler correctness proofs using proof systems includes Sokolowski's LCF work [163]. Using HOL Joyce has verified a compiler with as target machine a non-idealised formally verified computer Tamarack [82] taking into account finite storage [107]. A group at Computational Logic has used the Boyer-Moore theorem prover to verify a code generator [187], assembler, and linker [133] to a verified microprocessor FM8502 [98,99].

Embeddings of Hardware Description Notations

Brock and Hunt [22] describe a simple hardware description language in the Boyer-Moore theorem prover. It lacks delays and does not permit recursion; it thus deals with combinatorial logic only. However, this is the earliest research known to us which defines an operational semantics for an HDL in a proof system. Circuits are encoded as list constants, which are interpreted by a semantic function. For example, a full adder is described as follows.

```
'(half-adder (a b) (sum carry) (((sum) (b-xor a b))
                                 ((carry) (b-and a b)))
'(full-adder (a b c ) (sum carry)
            (((sum1 carry1) (half-adder a b))
             ((sum carry2)  (half-adder sum1 c))
             ((carry)       (b-or carry1 carry2))))
```

The circuit *half-adder* is defined as having two inputs a and b, and two outputs *sum* and *carry*. *b-xor* and *b-and* represent primitive XOR and AND gates respectively. The circuit descriptions may be hierarchically composed, as is seen in the full adder definition. A well-formedness predicate is defined to check that these definitions are purely combinatorial. The output value of a circuit description is computed by an operational semantics. It is encoded as a total recursive function; relational or partial definitions are not possible in the Boyer-Moore logic. The conceptual type of the semantic function is as follows: [5]

$$heval: \ name \rightarrow signalenv \rightarrow circuitenv \rightarrow value\ list$$

name consists of the name of the top-level component and its inputs. An environment *circuitenv* contains the definitions of non-primitive functions, such as *half-adder*. An environment *signalenv* is used to store the values of input, output

[5] The actual type is slightly more complicated because the function encodes two mutually recursive functions.

and internal variables such as *sum*. List literals may be used to great advantage here. To evaluate the half adder with inputs x and y with values F and T respectively, we use:

(heval '(half-adder x y) (list (cons 'x F) (cons 'y T)) (list '(half-adder (a b)...)))

The Lisp quote ' converts the variable x into a constant, which may be used as a name in the environment *valueenv*. (Recall that the HOL solution has been to formalise the type of names of wires to build an environment associating wires with their values (*str* in Equation 2.5.) In Section 4.2.2 we adopt the de Bruijn encoding [48] to deal with this problem.) The semantic function traverses valid abstract syntax trees, computing subcircuit outputs as it goes along. The semantics of primitive components such as *b-xor* is hard-wired into the interpreter. Other, non-standard semantic functions are also defined, computing the delay, and size (in number of primitive gates) of circuit descriptions. A recent extension to this work allows sequential circuits with delayed feedback loops [23]. State-holding components are listed explicitly in circuit descriptions.

Camilleri has given a semantics for CSP [96] in HOL [32]. The semantics is denotational; an abstract data type of CSP terms is mapped onto objects of type process. To reason about CSP terms one must obtain their corresponding processes from the semantics, and then reason about these processes.

To our knowledge our work is the first to describe the embedding of a semantics for a (subset of an) industrial HDL [69,72]. The HDL contains unit delays, generalised multiplexors and allows recursion. Both delayed and delayless feedback loops are allowed. We use the LAMBDA proof assistant, described in Chapter 4. We define a data type to define the abstract syntax of the HDL. As LAMBDA does not allow recursive relational definitions, the operational semantics is given as a function which is defined structurally on abstract syntax terms. This limits proofs to structural induction on program terms. Unlike the Boyer-Moore research, we do not need explicit environments (*circuitenv*, above) to bind circuit names such as *'half-adder* to their defining abstract syntax term. We use LAMBDA's logical infrastructure (discussed in detail in Section 4.1) to define abbreviations for sub-circuits.

van Tassel's VHDL embedding in HOL [175] uses the approach described in this thesis rather than the axiomatic and denotational semantics previously used in HOL. He gives an operational semantics for a small subset of VHDL, which deals elegantly with VHDL's problematic delta time. Delta time is used to represent the underlying simulator model's iterative behaviour. Any number of delta time clock ticks may occur within one discrete time interval. An abstract data type is used to represent program terms. A HOL package to define inductive relations [125] is then used to define a relational semantics. A rule induction principle for this set of rules is automatically derived by the package. This is a more general induction method than our structural induction on the abstract syntax, and the fixed point induction used by the LCF research described previously. We can only use functions to model semantics, restricting our semantics to functional semantics, where the HOL system allows more general relational semantics. The embedded semantics has been used to simulate simple circuits, in the manner discussed in Section 5.1. However, we are not aware of any results about the embedded semantics itself, such as totality. For example, a characterisation of circuits for which the operational semantics terminates would be very useful.

2.4 Conclusions

In this chapter we have argued for the separation of the description of structure and behaviour of hardware. We have seen how HDLs and proof systems may be combined, starting by extracting the behaviour of circuits directly. Various stages of automation using proof systems were described. The derivation of the behaviour via a formal semantics, and the progressive use of proof systems were reviewed also. The use of a formal semantics for an HDL embedded in a proof system allows powerful manipulation of both circuit descriptions and semantics. A number of operations such as symbolic simulation, circuit synthesis, and circuit optimisations can be used within one framework when using this approach.

Chapter 3

The picoELLA Language and Its Semantics

In previous chapters we presented the case for combining a hardware description language and a proof system by embedding the HDL's formal semantics. Here we present a small subset of a commercial HDL and its static and dynamic semantics.

3.1 Choosing an HDL

In this section we consider two candidate HDLs in detail: ELLA [45,137] and the VHSIC hardware description language VHDL [101,161]. Other hardware description languages which have been formalised will be discussed briefly in the related work section. Both ELLA and VHDL have had governmental backing, and are widely used in industry. ELLA was designed by the Royal Signals and Radar Establishment (now Defence Research Agency) in Malvern, UK for the Ministry of Defence in the UK in 1978. It is the one of the 'preferred HDLs' of the MoD. The US Department of Defence commissioned VHDL in 1983. It was standardised by the Institute of Electrical and Electronic Engineers (IEEE) in 1987 [101]. We consider these two languages because a goal of this research is to promote our formal methodology to industry. Using a stable, widely used language facilitates this process. Fortunately, it turns out that ELLA has a mathematically well defined and elegant semantic model, first described by Elliot [52]. To compare ELLA and VHDL we will describe the philosophies behind both languages, and discuss their underlying simulator models.

3.1.1 VHDL

In this section we first give a high-level outline of VHDL, we then sketch the sort of data values permitted, followed by the main modes of hardware description. We then describe the simulator model for VHDL, and briefly discuss the disadvantages of this model. Finally we describe research into providing a formal model for VHDL. We omit a large number of features of VHDL for lack of space, and because we are more interested in the underlying mechanisms rather than the particulars of specific constructs.

The very high speed integrated circuit hardware description language (VHSIC HDL, or VHDL) was developed for the Department of Defence in the US in response to the need for a standard language to describe, document, and exchange hardware designs [161]. VHDL is intended to cover the whole of the design process, from high-level specification to the gate level. As a result it is a very large language, and includes many features which deal with programming in the large. Most of these constructs have been modelled on similar constructs in the Ada programming language [141,161]. Packages are a means to group declarations of types and subprograms (functions and procedures.) For example, a package could implement a three-valued boolean logic data type as a data type with associated operators. The *design entity* is the fundamental concept for hardware description in VHDL. Design entity declarations describe input and output ports of a hardware component. Assertions about behaviour may also be included. For every design entity declaration there may be one or more implementations, called architecture declarations. In a final configuration of a hardware design every design entity which has been used must be instantiated with a particular implementation.

Type definitions in VHDL resemble closely those of Ada. Enumerated types, arrays, records, various kinds of floating point and real numbers are supported. In the behavioural subset of VHDL pointers may also be used. Signals and physical types have been introduced to deal specifically with hardware design. The former represent the changing values of wires, the latter are used to define the time scale, for example.

There are two principal modes of description for architecture declarations: behavioural and structural. A third form, data flow, may also be used, but is a shorthand for a behavioural description. To illustrate different styles of descriptions in VHDL, we will use a two bit adder as a running example. Here is a possible interface declaration.

```
use BitPackage.all;
entity TwoBitAdder is
       port (in:  in bit (0 to 1);
             carryin:  in bit;
             sum:  out bit (0 to 1);
             carryout:  out bit);
end TwoBitAdder;
```

The `use` clause specifies that the declarations of the package `BitPackage`, such as type `bit`, are included. A design entity may be composed of smaller components in a purely structural manner. Leaf components of this structural hierarchy will be behavioural descriptions. Structural descriptions cannot be leaf components because they define no behaviour; they can only compose behaviours of subcomponents. Moreover, behavioural and data flow descriptions may not contain structural descriptions. This means that behavioural descriptions can only be decomposed behaviourally, or else a corresponding structural description has to be provided. A two bit adder could be described in a structural manner using full adder subcomponents as follows.

```
architecture Structural of TwoBitAdder is
             component Adder
             port (ina:  in bit;
                   inb:  in bit;
                   sum:  out bit;
                   carry:  out bit);
             end component;
             signal carries :  bit (0 to 1);
begin
             Add0:  Adder port map (in (0), carryin,
                                    sum (0), carries (0));
             for i in 1 to 1 generate
                 Add:  Adder port map (in (i), carries (i-1),
                                       sum (i), carries (i));
             end generate;
             carryout <= carries (1);
end Structural;
```

This particular implementation, or architecture, is called `Structural`, and it contains a one bit adder subcomponent called `Adder`. In a final configuration we have to specify the particular implementations which will be used for the two `Adder`s. Here we only know the interface of the `Adder` entity as described in the port map. Ports of subcomponents are connected using signals `sum` and `carries`. `sum` and `carryout` have been declared in the interface declaration `TwoBitAdder`. Particular subcomponents may be named using labels, such as `Add0`. The structural replication (`for i in ...`) which we have used may be used to a more obvious advantage in larger or parametrised adders.

A behavioural description of the two bit adder is given below.

```
architecture Behavioural of TwoBitAdder is
            function carryfunction (x: bit (0 to 1); y: bit) ...;
            function sumfunction (x: bit (0 to 1); y: bit) ...;
begin
            process begin
                    sum <= sumfunction (in, carryin);
                    carryout <= carryfunction (in, carryin);
                    wait on in, carryin;
            end process;
end Behavioural;
```

`sumfunction` and `carryfunction` are functions with a bit array and bit input, and return a bit array and bit respectively. They are conventional programming language programs, and there is no indication of any particular hardware implementation. The process statement models hardware as a continuously running sequential program. After the output signals `sum` and `carry` are assigned the appropriate values, the process waits until a new value arrives on either of the two input signals `in` or `carryin`. Waits for signal changes, time-outs, *etc.* must be explicitly specified using the `wait` statement. In a top-down design methodology, the first implementation of a design entity would be a high-level behavioural description. Subsequent refinements would partition the behavioural description into smaller components. The behavioural style most clearly shows the underlying simulation model of VHDL, which we will describe shortly.

The third description style, data flow description, is intended to be similar to a register transfer description style.

```
architecture DataFlow of TwoBitAdder is
begin
            sum(0) <= in(0) and carryin;
            ...
            carry <= in(0) xor (in(1) xor in(2));
end DataFlow;
```

Descriptions using the data flow style are syntactic sugaring for more explicit behavioural descriptions. Every data flow (or concurrent) signal assignment is equivalent to a process containing the signal assignment, followed by a wait statement on the signals in the right hand side of the signal assignment. In the remainder of this section we will assume that data flow descriptions have been rewritten to their corresponding behavioural form.

VHDL's Simulation Model

The following description of VHDL's simulation model [101, Section 12.6] is necessarily incomplete. We try to convey an intuition for the basic behavioural model, but ignore more advanced features. We briefly mention how signals are resolved over the structural hierarchy.

Every design is a structural decomposition containing behavioural descriptions at the leaves. These may be composed in a structural or behavioural manner. VHDL's simulation model therefore has two parts: the first deals with the structural aspects of the design, the second with the behavioural elements of one design entity. The latter is more fundamental, and we will discuss it first.

A behavioural description consists of a number of communicating sequential processes. The computation model is designed to deliver the same result irrespective of the execution order of the processes in a design entity. This is achieved as follows. Each process is a sequential program which may communicate with other processes only via signals. Variables are strictly local to a process, and may not be used for interprocess communication. Processes are perceived as running in parallel, synchronising occasionally to update shared signals. An unresolved signal may be read by any number of processes, but only written by one process. A resolved signal may be written and read by any number of processes. Resolved signals may be used to model buses. In order to ensure that all reading processes read the same value of a signal, an assignment to a signal is not effected until all processes have synchronised. (This may also be interpreted as a buffering of signal assignments.) Synchronisation is explicit, using wait statements. Processes have synchronised when they are all suspended, *i.e.* waiting for a change in a signal. If this is the case, the values of all signals which have been assigned to are updated. For resolved signals this entails computing the value from the multiple assignments to the signal using a resolution function. (Or: compute the global value using the buffered values.) A resolution function could represent a bus resolution algorithm, a hard-wired OR, or any other protocol. At this point the global store has been updated and all processes will read the new signal values. All processes are then restarted. Of course, in an actual implementation of the simulation algorithm it is unnecessary to rerun a process if it does not depend on (is sensitive to) any of the signals which have changed. This process continues until no signals have been changed. The period between two process synchronisations is called a simulation step. Individual simulation steps are not accessible to the user, but their effects are visible. A simulation step takes one so-called 'delta time.' Only when all processes have synchronised, signals have been updated and there are no more signal changes, is time advanced. Zero or more delta time steps take place in every time step, which the user may use explicitly. There may be an infinite number of delta time steps in a time step, which means that the simulation does not terminate. VHDL's hierarchical timing model is derived from that of CONLAN [147]. The simulation model is given in a pictorial form on the next page.

We will not give any details about the part of the simulation model which deals with inter-design entity communication. Input and output ports of a design entity behave 'as expected' when they are not connected to resolved signals. The value of resolved signals, however, is computed at each level of the hierarchy, and passed up through a port. If the port is an `inout` port, the final resolved value is passed down again. A resolved signal may therefore have two different values: one which is passed up through output ports, and another which is received through input ports. The model is further complicated by the possibility of conversion functions which operate at input and output ports.

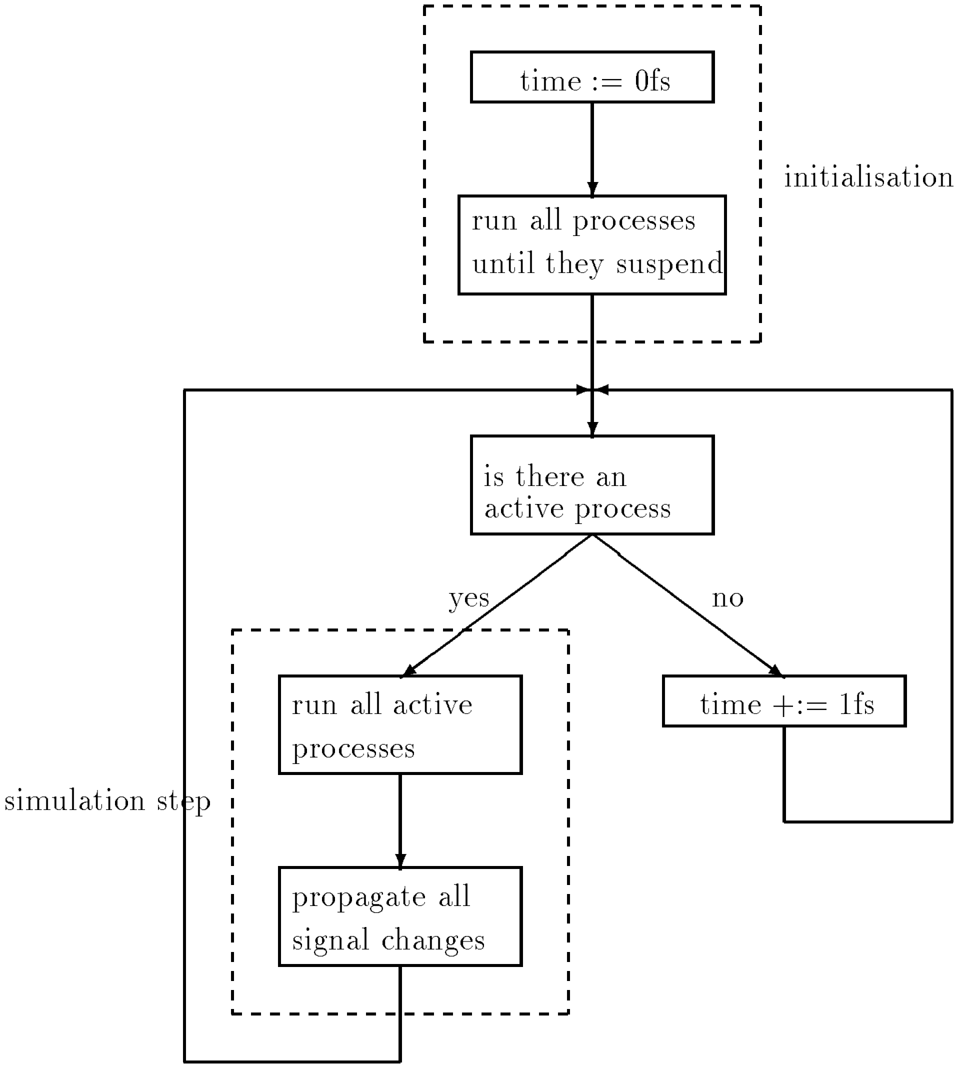

Figure 3–1: The VHDL Simulation Model

To clarify this model consider the following example.

```
use BitPackage.all;

entity Example is

       port (input:  in bit; output:  out bit := '1');

end Example

-- continued overleaf
```

```
architecture Eg of Example is
begin
            NAND: process begin
                          output <= not (input and output);
                          wait on input, output;
            end process;
end Eg;
```

`output` is initialised to `1`. If `input` is `0` then the `NAND` process will assign `1` to `output`. The process then waits for a signal change on `input` or `output`. There are no other processes to finish so we can conclude this simulation step, because the signal `output` has not changed. It therefore takes one simulation step to converge to a solution, and time advances. If on the other hand, `input` is `1`, `output` is first assigned `0`. The process then waits for a signal change on `input` or `output`. There are no other processes to finish so we resume, because the signal `output` has changed from `1` to `0`. The second time around we assign `1` again to `output`. We suspend and resume again, this time to assign `0` to `output`. These last two simulation steps will be repeated *ad infinitum*. The circuit fails to converge to a solution, and time does not advance.

As the preceding exposition will have made clear, VHDL is a large language with an intricate simulation model. The user must be familiar with this operational model before the language can be used. A simulation must be understood at this level because the simulator mechanism affects the behaviour of circuit descriptions at a high level. Consider the following illustrative VHDL fragment.

```
process begin
        s <= '1';
        assert s = '1' report "s <> 1" severity error;
end process;
```

If `s` is `0` before the signal assignment, the assertion will fail because signal assignments are buffered. Explicit delays may be introduced at signal assignments. For example, `s <= transport '1' after 10fs;` assigns `1` to the signal `s` 10 femtoseconds after the current time. VHDL documentation describes this as *scheduling a transaction*, which confirms the event-driven nature of the simulation algorithm. Inertial delays are also provided by VHDL. Their definition is given in terms of event queue manipulation [101, Section 8.3.1]. Thus the VHDL timing model is forward looking in the sense that future values of a signal are manipulated.

The simulation model has the feel of a conventional programming language. A number of sequential subprograms execute in parallel, communicating using a rather unintuitive synchronisation mechanism to ensure that the execution order of the processes does not matter. Indeed, when a programming language such as Ada or C is used as an HDL, a similar style of hardware description emerges [3,67,111]. Not only is the simulation model visible, it may be used in hardware descriptions using signal attributes, as demonstrated below.

```
process begin
        wait on input;
        if input = '1' then
                s <= '1';
        end if;
end process;
process begin
        s_is_active <= s'active;
end process;
```

Here we assign `true` to signal `s_is_active` if signal `s` has been *active* in the current simulation cycle. A signal is active if it has been assigned to, but its value need not have changed. This makes no sense from a hardware point of view. Although the above description looks plausible enough, an implementation would have to model the simulation algorithm in hardware to compute the `active` attribute. The fact that the simulation model is event-driven should be irrelevant to any hardware description. However, the current situation allows circuit descriptions which rely explicitly on the event-driven simulation algorithm.

Research into Formal Semantics for VHDL

The complication of VHDL's simulation model has slowed down progress on a formal semantics. Borrione's group in Grenoble has been working on a functional model for VHDL [14,16,157]. The PREVAIL proof environment [16] translates a subset of VHDL which describes synchronous sequential circuitry into the Boyer-Moore theorem prover. Tail recursive functions are used to model the passing of time, and their arguments represent the state. For zero delay combinatorial hardware (a non-recursive functional model in) the TACHE theorem prover is used. Salem has done some initial work towards reasoning about VHDL timing constructs using the Boyer-Moore theorem prover [157]. He proves some lemmas involving signal attributes which are stated without justification in the VHDL reference manual [101]. His work differs from other VHDL semantics in the use of a backward looking timing model rather than the standard preemptive signal semantics. In both research efforts multiple processes without resolved signal assignments were discussed. It is hard to see how delta delays may be incorporated into the coarser (in effect unit delay) time scale supported by their Boyer-Moore model.

At the Aerospace Corporation Fillipenko *et al.* use the state delta verification system (SDVS) [56,114,113]. SDVS has been used to reason formally about Ada, ISPS, and VHDL. SDVS implements a first-order logic extended with temporal operators. State deltas are the basis for describing temporal properties. State deltas are rules which fire when their preconditions are satisfied. Postconditions specify their effect on the state. The semantics of the three languages has been given by behaviour functions mapping programs into state deltas, as described in the previous chapter. The subset of VHDL which can be translated into SDVS contains multiple processes, although only unresolved signals are allowed. A *four*-level timing hierarchy consisting is used to model VHDL's timing model. In increasing granularity these are: user-visible time (in femtoseconds), delta time, zero time and state delta time. We have described the first two previously. Zero time models the evaluation of sequential statements in a process, which may take several steps in the primitive (state delta) time scale of SDVS, but within one delta time step. An evaluation of a single statement may take more than one step on the primitive state delta time scale.

In [174,176] van Tassel describes how the sequential subset of VHDL may be translated into the VHDL annotation language VAL, and into HOL. This work implements a behaviour function, as described in the previous chapter. More recently his HOL semantics [175] has taken the embedded operational semantics approach advocated in this thesis. This work has been reviewed in Section 2.3.1.

Wilsey has taken a much wider look at VHDL by including design entities and inter-architectural communication which has been ignored in all other work [181, 182]. Resolution functions, type conversions, and unrestricted wait statements are all included. This research is also distinctive in its use of temporal interval logic to specify the behaviour of signals.

Umbreit [172] translates VHDL programs into ML programs which may be reasoned about using the LAMBDA proof assistant. The generated ML programs are simplified automatically. In effect, a simulator which is instantiated with a particular program is generated. Thus no general facts can be proved about the simulator, only about particular instantiations. A large subset of VHDL, including functions, multiple processes, signal assignments, *etc.* is treated. However, delta delays are excluded.

Other related work includes [39].

3.1.2 ELLA

ELLA[1] [45,137] is very different from VHDL. It is a relatively small language with a fairly simple simulator model. Few concepts suffice to make it very powerful due to its orthogonal design. In contrast to VHDL, ELLA does not provide support for designing in the large directly. This is handled by the ELLA applications support environment EASE instead [144]. The basic design unit is the function. Different implementations of the same function are handled by using different contexts in EASE.

Conceptually, a circuit is a network of interconnected nodes. Functions take the place of the nodes, and function applications or explicit joins represent the arcs. This model may be hierarchical; nodes can contain subnetworks. All nodes are thought of as operating at all times, and operate in zero time. Delays must be introduced using explicit delay nodes. ELLA contains few primitive components: a generalised multiplexor or truth table construct, various types of delays, `RAM`, and `BIOP` statements. `BIOP` constructs are operators dealing with built-in data types such as integers, reals and strings. All other ELLA constructs may be translated automatically into this subset. All recent extensions to ELLA have been defined in this manner, *cf.* [94].

Data types supported by ELLA include enumerated types, tuple types which subsume arrays and unnamed records (rows and collaterals in ELLA parlance), and function types. In ELLA version 4.0 strings and reals were introduced. Enumerated types may be simple enumerated types or associated types. The latter allow a value to be associated with elements of an enumerated type. For example, consider the following enumerated and associated types.

[1] We used ELLA version 3.0 as the basis for our work. For this reason we cite the version 3.0 manual [151] when we wish to illustrate a particular point. The version 4.0 manual [45] will be referenced in general contexts. Apart from some new features the only difference of interest between version 3.0 and 4.0 is explained on page 56.

```
TYPE bit = NEW (hi | lo).                      # enumerated type #
TYPE address = NEW addr(0..255).               # ELLA integer type #
TYPE addrdata = (address,[8]bit).              # address and 8 bits #
TYPE enable = NEW (read & address | write & addrdata).
                                               # associated type #
TYPE bustype = enable -> [8]bit.               # function type #
```

Replicators such as `[8]` define an array of elements. Elements of type `enable` can be either a `read` with an associated address, or a `write` with associated address and datum. The last type is a function type, but no data values of this type exist. It is a purely syntactical device to deal with buses and plug-in components. All types but function types implicitly declare a 'don't know' element. This treatment ensures that a consistent approach exists for initialising delays *etc.* with unspecified values, and during simulation allows irrelevant inputs to be designated as such. All types contain a finite number of elements. Thus the built-in integer and real types have a predefined, rather than a *de facto*, limited range.

Three modes of hardware description are supported: an explicit net-list style, a functional or implicit net-list style, and a sequential style. We will give an example of each of these styles below. A major difference with VHDL is that these three styles may be mixed arbitrarily. This is due to the consistent treatment of expressions; wherever an expression is expected any program fragment which delivers an expression is allowed. As mentioned above, all basic hardware descriptions are delayless; delays have to be introduced explicitly using delay nodes. This makes designs more verbose, but also highlights where the design depends on certain timing conditions. In VHDL it is possible to describe a flip-flop constructed from two cross-coupled NAND gates without the mention of any explicit delays. The simulator model ensures that, in this case, the design does not oscillate. This masks the fact that, if both NAND gates had zero delay or exactly the same delay the design would fail with certain inputs. In ELLA the same description would output undefined for these inputs unless explicit delays were introduced. The function

`DEL` is a pure delay of one time step and its input is a two bit vector.

```
FN DEL ([2]bit:  in) -> [2]bit:  DELAY ((lo,lo), 1).
```

The output of the delay at time zero is `(lo,lo)`.

As an example of the functional description style consider a full adder.

```
FN ADDER = (bit:  x y c) -> [2]bit:  # sum, carry #
BEGIN
     LET xor = XOR (x,y).  # fan-out of signal #
OUTPUT DEL ((x AND y) OR (c AND xor), xor XOR c)
END.
```

All subcomponents are used as functions operating on expressions. Every application of a function generates a distinct piece of hardware, so that there are two AND gates in the above description. The output of the `XOR (x,y)` expression is given an explicit name because we want a fan-out of the signal. Thus there are two XOR gates in the adder. If we substituted `XOR (x,y)` for the two occurrences of `xor` we would obtain an implementation with three XOR components. The adder has a one-unit delay, expressed by the final call to `DEL`.

A two bit adder may be described in a structural, or explicit net-list style as follows.

```
INT n = 2.
FN NBITADDER = ([n]bit:  in, bit:  carryin) -> ([n]bit, bit):
BEGIN
     MAKE [n]ADDER: adders.
     JOIN (in[1], carryin) -> adders[1].
     FOR INT i = 2..n
         JOIN (in[i], adders[i-1][2]) -> adders[i].
OUTPUT ([INT i=1..n] adders[i][1], adders[n][2])
END.
```

This description is parametrised on the number of bits. It is not as general as it could be, but is given as a comparison with the VHDL fragment on page 42. `n` is a constant, but it is possible to instantiate `NBITADDER` with different `n` by making the function definition generic on `n`. Subcomponents are declared explicitly using the `MAKE` statement, and are connected using the `JOIN` statements. To duplicate something in ELLA it may be prefixed with a replicator `[n]` for a number of identical copies, or `[FOR i=n1..n2]` to vary the expression with the index `i`. These constructs may be applied to any expression, which makes them easy to use.

Finally, behavioural descriptions are supported in ELLA through sequential statements. These are more limited than the behavioural subset of VHDL. All ELLA descriptions have a hardware intuition [138]. That is, every circuit design may be expanded to a structural or explicit net-list description. Of course, the values operated on by this description need not be as low-level as bits say. Consider an RS flipflop:

```
FN RSFF = (bool:  set reset) -> bool:
BEGIN SEQ
     VAR state INIT x;
     CASE (set, reset) OF
     (lo,hi):  state := f,
     (hi,lo):  state := t,
     (lo,lo):  # no change #
     ELSE state := x
     ESAC;
OUTPUT state
END.
```

In this imperative style the variable `state` retains its value from one time step to the next. Also consider `BTOD` which converts an `n` bit vector to an integer.

```
MAC BTOD {INT n} = ([n]bit:  vector) -> int:
BEGIN SEQ
     VAR answer := i/0;
     [INT j = 1..n]
          answer := (answer * i/2) + ABS vector[j];
END.
```

Parametrised functions, or macros, can use replicators, or use structural recursion. They may be parametrised on types, integers (*e.g.* word size), functions, constants, *etc.*

ELLA's Simulation Model

The version 4.0 ELLA simulation model [45] is very simple. Recall that all circuits may be considered as a network of interconnected nodes. We create such a network by expanding all non-primitive ELLA constructs, such as sequences and function types, and creating nodes for all implicit function calls. At every time step we initialise the outputs of all functions to the undefined or don't know value of the appropriate type. We repeatedly compute all function outputs in any order. The re-evaluation of all functions at every pass corresponds to the notion that all hardware runs continuously. Time is advanced when all function outputs have stabilised. Of course, in the absence of delayless feedbacks, this reduces to a single pass computation, which follows the flow of data. Delays output a constant value during one time step, so that feedbacks through a delay are no problem. Function outputs do not need to be initialised to the undefined value in this case.

When delayless feedbacks are introduced, more than one pass may be necessary, but the circuit will stabilise within a finite number of steps. It may be proved that this always happens (Section 4.3.) The proof of this property depends crucially on the presence of the undefined value implicitly defined for every type. This value introduces a 'definedness' ordering on values of a type. The don't know value (*e.g.* `(?bit,?bit)`) is least defined or most pessimistic, because we know nothing about it. A value such as `(hi,?bit)` is partially defined, and thus larger than `(?bit,?bit)`. `(hi,lo)` is fully defined, and therefore larger than `(hi,?bit)`. In an enumerated type all constructors are larger than the undefined value, but incomparable to one another. This is a *flat data ordering*. We can then extend this ordering to tuples and associated types. Using this ordering we can show that every construct in ELLA is monotone. That is, if we obtain more information about an input we also get to know more about its output. (Strictly speaking, we know at least as much.) The basic totality proof is as follows. At every time step we start with the most pessimistic estimate possible because all outputs are set to undefined. If we evaluate the functions in any order they either use an old, pessimistic estimate, or an up-to-date estimate. By repeatedly computing all functions, an up-to-date, *i.e.* least pessimistic value percolates through the network, following the flow of data. The monotonicity of the nodes is crucial, because it allows delayless feedback loops to increase the definedness of their outputs using previous pessimistic estimates. As we allow only a finite number of functions, and all types contain only a finite number of elements we cannot find more defined estimates forever. Due to the monotonicity of ELLA constructs the output cannot become less defined, and because there is an upper bound on the number of elements in a type it follows that we neither decrease nor increase our current estimate. In other words, we have arrived at a fixed point. In particular we have computed the *least fixed point* solution because we started with the most pessimistic estimate. There may be other fixed points, but these cannot be reached in this manner. This fixed point model for ELLA was first described by Elliot [52].

The version 3.0 ELLA [151] simulator did not deal properly with delayless feedback loops. The feedback wires were not initialised to the don't know value, but used the output from the previous simulation time instead. This could lead to infinite loops in the simulation. The user had to provide an upper bound of the number of iterations to prevent this.

The fact that the simulation model is, in both cases, implemented in an event-driven rather than iterative manner is completely irrelevant to the user, as long as the implementation exhibits the 'official' behaviour. There is no way in which the user can access the events in the simulator. It is not possible therefore to write an ELLA function corresponding to the VHDL fragment on page 48. However, one may write functions which give the same information as VHDL attributes such as `'stable` which have a hardware intuition. These functions are directly implementable in hardware, by virtue of the direct mapping of ELLA onto hardware. Thus there is no need to emulate ELLA's simulator mechanism in hardware to be able to do this.

3.1.3 ELLA versus VHDL

VHDL is a broad-spectrum language; it may be used from very high-level algorithmic designs down to transistor-level circuit descriptions [109]. Although ELLA is designed to be used from the gate level upwards, it may be used to model transistor-level circuitry. However, this becomes quite tedious and convoluted. At the higher levels, behavioural descriptions may not be as good as those in VHDL, due to the lack of imperative programming language features. Designers therefore tend to use ELLA in the middle ground [137,180]. In general VHDL is a richer language than ELLA. For our purposes this is a disadvantage as we need a tractable rather than a complete language. The ability to translate ELLA into a small subset is a definite advantage. ELLA's orthogonality and consistent treatment of its constructs was more suitable to the approach where a subset of the language was given a semantics followed by a derivation of the semantics for the remainder. In VHDL this is harder due to a larger number of primitive concepts, and their distinct treatment (*cf.* structural versus behavioural implementations.)

The most important difference from our point of view is the simplicity of the simulation model. ELLA's evaluation model is simple and mathematically pleasing, whereas the same cannot be said for VHDL. Although both ELLA and VHDL have a concept of simulation cycles within one time step, only in ELLA's case are these invisible and inaccessible to the user.

Overall, we preferred the relatively simple ELLA with a well-defined core of constructs which needs more explicit user-guidance in the case of timing to a more complete VHDL which tries to anticipate users' needs by using a more complicated simulator model.

We devote the remainder of this chapter to the description and definition of picoELLA, the subset of ELLA we will be working with. Its formal static and dynamic semantics will then be presented, followed by a discussion of research related to subsets of ELLA and their semantics.

3.2 A picoELLA Rationale

In this section we first highlight some inadequacies in the ELLA reference manual which indicate a need for a formal semantics. Following this, we define and justify a minimal subset picoELLA which we will be using.

The need for a formal semantics becomes clear when we see how the ELLA reference manual [151] describes the operation of the sequential case clause:

> The `CASE` clause is similar to the `CASE` clause described in Chapter 7, but the arms of the `CASE` clause do not deliver a value. This means that a colon need not be followed by a 'statement.'
> [151, Section 9.3.4.5]

The sequential case clause is not value delivering, so how are we to reconcile this with the definition in Chapter 7:

> If the `ELSE` part is omitted and the input does not match any of the 'choices', the output of the `CASE` clause is an unspecified value of the appropriate type ('?').
> [151, Section 7.3.3.1]

A strict interpretation would be to abort the simulation because the answer is not defined. The simulator gives a more optimistic but safe answer[2]: all signals depending on the `CASE` statement are set to undefined. This is an example of a construct which is translated into simpler ELLA and so given a derived semantics [138]. The description of the behaviour of a value delivering `CASE` statement when undefined values are present [151, Section 7.3.3.2] is another area where a formal definition would be beneficial. Hill *et al.* [93] provide a clear and unambiguous definition of the unsatisfactory 'definition by example' of ELLA delays in [151, Appendix A.3].

In [70] we described a formal dynamic operational semantics for an ELLA subset called microELLA. It is the only attempt we are aware of which deals directly with ELLA's sequential constructs. Their semantics turned out to be very inelegant, because an imperative style of hardware description did not mix well with the functional evaluation model. This semantics followed closely the simulator model as described above. This resulted in a very imperative style of semantics. All function outputs were stored explicitly in a store, or memory. This store was carried over from one time step to the next to model delays. This work highlighted the need for a clean semantics, and showed that a minimal subset into which the remainder of ELLA was to be translated was the correct way to proceed.

We concentrated on finding the smallest subset of ELLA which would still have all its salient features. We called it picoELLA [71], and it is described below. We later learned that Davies, at the Royal Signal and Radar Establishment, had previously arrived at a nearly identical ELLA subset [47].[3] In the remainder of this section we will describe picoELLA and the justification for its constructs. The next section gives a formal static and dynamic semantics for picoELLA. Research involving various subsets of ELLA, including that of Davies, will be reviewed in

[2]Safe in the sense that we cannot construct non-monotone circuits.

[3]Davies' language did not include indexing, and `LET` statements, which are included in picoELLA for pragmatic reasons.

Section 3.3.6.

picoELLA Constructs

The basic components in ELLA are the `CASE` statement (without `ELSEOF`), `DELAY`, `IDELAY`, `RAM`, and `BIOP` statements [136]. Expressions may be bundled into tuples, and parts extracted using indexing. From our point of view two different types of delay are redundant, as we are only interested in the principle of a delay. For practical purposes built-in operators are important, but they do not add any new theoretical insights (but see pages 227–229.)

ELLA's model of hardware consists of a hierarchical network of nodes connected by a number of arcs. By flattening out this network by removing all functions, we are left with a net-list which consists of basic ELLA components. This network may be described by a single functional expression, if feedback is represented by recursive `LET` constructs. From a semantic point of view we can dispense with fan-outs, or non-recursive `LET` statements, by replacing every occurrence of the variable by the defining expression. However, this is not the same from the hardware point of view, because in that case we replicate hardware instead of sharing it. We therefore include the `LET` statement. We have no need for functions, because they have all been flattened out. We would like to stress that we are concerned with the least number of semantic constructs rather than ease of circuit description. Having said that, for pragmatic reasons we include indexing operators, which strictly speaking are superfluous. However, without indexing it becomes hard to combine program fragments, which we will be manipulating more often than complete programs.

Enumerated types, including associated types, and tuple types are the only essential types. In conjunction with the implicitly declared undefined or bottom element (non-associated) enumerated types generate a flat data ordering. Tuple types form a product data ordering and associated types a tagged union data ordering [52] (figures 3–2,3–3.)

```
TYPE enum = NEW (true | false).
TYPE tuple = (enum,enum).
TYPE assoc = NEW (this & enum | that & enum | empty).
```

`enum` consists of three elements: `true`, `false` and `?enum`. The bottom element `?tuple` is equal to `(?enum,?enum)`. The data orderings may be visualised as the following semi-lattices.

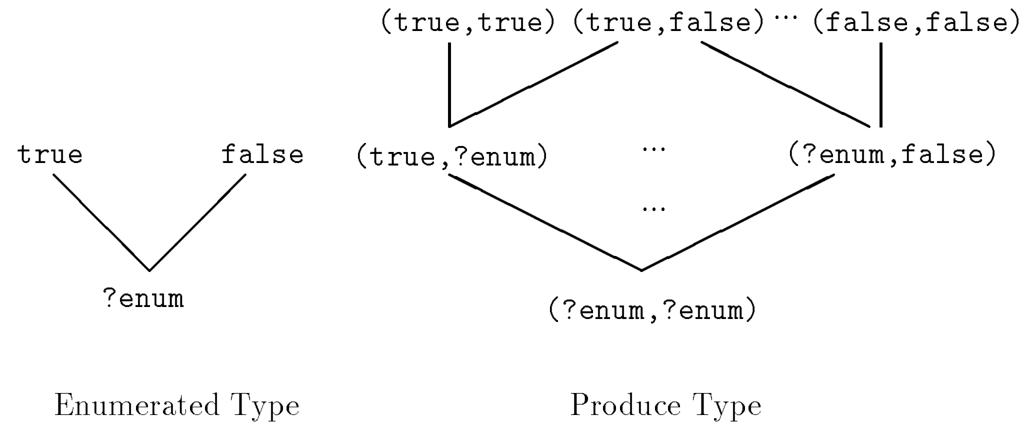

Figure 3–2: picoELLA Flat and Product Data Orderings.

We omitted a number of elements, indicated by ellipses. The associated type contains the following elements: `?assoc`, `empty`, `this&?enum`, `this&true`, `this&false`, `that&?enum`, `that&true`, `that&false`.

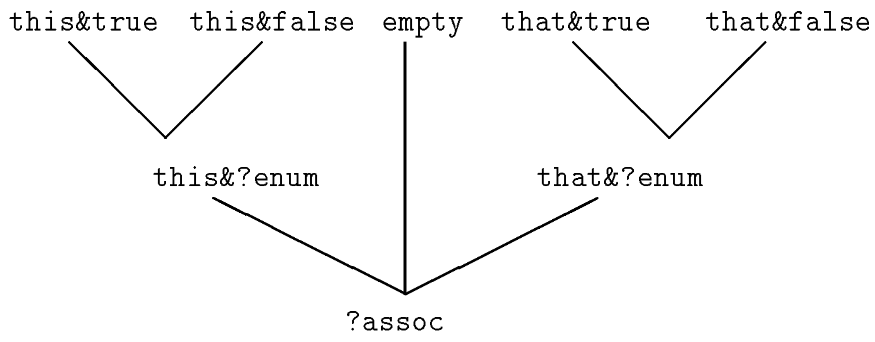

Figure 3–3: picoELLA Tagged Union Data Ordering.

Although associated types introduce a new kind of data ordering we decided to omit them from picoELLA because basic enumerated types and tuples provided enough interest. Moreover, associated types can be encoded using tuple types. `assoc` can be represented by the following type `simassoc`.

```
TYPE three = NEW (this | that | empty).
TYPE simassoc = (three,enum).
```

However, assuming `(empty,?enum)` encodes `empty`, `simassoc` contains some redundant elements such as `(empty,true)` and `(empty,false)`.

Given the above considerations we included the following constructs in picoELLA: type definitions, local declarations, recursive declarations, constants, tuples, indexing, a simplified `CASE` statement and a delay. The BNF of picoELLA is given on page 66, but an informal description will be given first. Notational conventions for picoELLA are: all keywords are given in `CAPITALS`, all constructors and type names in `teletype` font, and all variables in *italic*. This agrees with notation in later chapters.

Declarations

Type definitions may declare either a finite number of constructors of an enumerated type, or define a binary tuple type using previously defined types.

```
TYPE bool = true | false IN
TYPE bit = hi | lo | x | z IN
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. We will assume throughout that an unlimited fan-out is permitted, although it is possible to limit fan-out.

```
LET double = expr IN (double,double)
```

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 INIT ?bit REC out = DELAY (hi, IF out MATCHES hi
                                          THEN lo ELSE hi)
IN out
```

The unusual term `INIT ?bit` uniquely types the expressions, and is also used in the computation of the least fixed point solution.

Expressions

Constants are either constructors such as `hi` from an enumerated type, a bottom value from a type, *e.g.* `?bit` or `?twobool`, or a constant binary tuple. Constant tuples can only contain other constants. Constants are used as initial values in delays and recursive declarations but may also be coerced to expressions.

Any two expressions may be combined in a binary tuple. Note that `(hi,hi)` is either a constant tuple coerced to an expression, or two coerced constants combined in an (expression) tuple. We always indicate which form is used in a particular context. An expression which has a tuple type (*i.e.* evaluates to a constant tuple) may be indexed. Tuples and indexing behave as usual; for example `((`x`,`y`)[2], (`x`,`y`)[1])` is equal to `(`y`,`x`)`.

`IF` statements are a simplified form of `CASE` statements. A `CASE` statement can have any number of branches and an `ELSE` part [151, Section 7.3.3]. Every branch has a chooser, which is a pattern indicating when that branch will be selected. Choosers must be disjoint to ensure a unique output. We have simplified this to a series of nested `IF` statements each containing a single chooser. Thus

```
CASE expr OF
(hi|lo,lo): lo,
(lo,hi): hi
ELSE ?bit ESAC
```

is encoded as

```
LET tmp = expr IN
IF tmp MATCHES (hi|lo,lo) THEN lo ELSE
IF tmp MATCHES (lo,hi) THEN hi ELSE ?bit
```

The `LET` forces a fan-out rather than a duplication of *expr*. Apart from the fact that we do not have to consider arbitrary length (chooser, expression) pairs we can also ignore the static type checking of the disjointness of all choosers. Choosers are constructed from (i) fully defined constants, (ii) wild card choosers, (iii) bar '`|`' choosers, and (iv) tuple choosers. It is not possible to match for partially defined values, as this would allow non-monotone circuit descriptions. Every type has a corresponding wild card chooser, *e.g.* `bit`, `twobool`, which matches all values of this type, including the bottom value. The bar '`|`' denotes disjunction: a value matches *ch*`|`*ch'* if it matches at least one of *ch* and *ch'*. This presents no problems because choosers cannot contain variables which may be used in the expression, as happens in ML-style pattern matching [90]. A tuple chooser represents pairing: it matches if both components match. Thus the chooser `(hi,lo)` can either be a constant tuple chooser, or a tuple chooser of two constant choosers. `(hi|lo,lo)` can only be a tuple chooser. The way in which this matching process is defined for bottom values is crucial. 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. This may be described best using an example. Take the following definition of an AND gate:

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

Another, equivalent, description is

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

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 final construct in picoELLA is the delay. It introduces a discrete linear time starting at time zero. `DELAY(`c_t`,` *expr*`)` denotes a unit delay with the output of circuit *expr* as its input. `DELAY(`c_t`,` *expr*`)` outputs the result of *expr* 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 *expr* 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. The type of the evaluation (as a function) within one time step is therefore $(\mathit{environment} \times \mathit{expression}) \rightarrow (\mathit{constant} \times \mathit{expression})$. Given a value environment and a circuit we output a value and a new circuit description, which will be evaluated at the next time step. The evaluation model for picoELLA is essentially the same as that for ELLA.

3.3 A picoELLA Semantics

The picoELLA semantics described here is formulated slightly differently from earlier versions, *e.g.* [71].

After the informal description of picoELLA constructs, above, we now define the formal syntax in the standard Backus-Naur Form.

[4] In ELLA the value in the description of a pure delay indicates not the state, but its initial value only.

$$
\begin{array}{rcl}
\textit{program} & ::= & \texttt{TYPE}\ \textit{tdecl}\ \texttt{IN}\ \textit{program}\ | \\
 & & \texttt{INPUT}\ \textit{name} : \textit{ttype}\ \texttt{IN}\ \textit{expr} \\
\textit{tdecl} & ::= & \textit{tname} = \textit{btype}\ \ |\ \ \textit{tname} = \textit{ttype} \\
\textit{btype} & ::= & \textit{cname}\ \ |\ \ \textit{btype}\,\texttt{|}\,\textit{btype} \\
\textit{ttype} & ::= & \textit{tname}\ \ |\ \ \textit{ttype} * \textit{ttype} \\
\textit{decl} & ::= & \texttt{INIT}\ \textit{const}\ \texttt{REC}\ \textit{name} = \textit{expr}\ \ | \\
 & & \textit{name} = \textit{expr} \\
\textit{expr} & ::= & \textit{const}\ \ |\ \ \textit{name}\ \ | \\
 & & \textit{expr}[\textit{int}]\ \ |\ \ (\textit{expr}, \textit{expr})\ \ | \\
 & & \texttt{DELAY}\ (\textit{const}, \textit{expr})\ \ | \\
 & & \texttt{IF}\ \textit{expr}\ \texttt{MATCHES}\ \textit{chooser}\ \texttt{THEN}\ \textit{expr}\ \texttt{ELSE}\ \textit{expr}\ \ | \\
 & & \texttt{LET}\ \textit{decl}\ \texttt{IN}\ \textit{expr} \\
\textit{chooser} & ::= & \textit{tname}\ \ |\ \ \textit{cname}\ \ |\ \ (\textit{chooser}, \textit{chooser})\ \ |\ \ \textit{chooser}\,\texttt{|}\,\textit{chooser} \\
\textit{const} & ::= & ?\textit{tname}\ \ |\ \ \textit{cname}\ \ |\ \ (\textit{const}, \textit{const}) \\
\textit{int} & ::= & 1\ \ |\ \ 2
\end{array}
$$

This BNF definition of picoELLA resembles that given in the version 3.0 ELLA reference manual [151, Appendix A.1].[5] In particular *cname* is part of two syntactic classes: *const* and *chooser*. Moreover, it may be coerced from a *const* to an *expr*. Assuming that coercing is explicit, it is always clear from the context which of the three classes *cname* belongs to. The reason we mention this here is that in the later versions of the ELLA syntax [45, Appendix A.1] and [136, Appendix B] constants and choosers are given as one syntax class *constset*. This syntactic nitpicking has a marked influence on how constants and choosers are perceived. This results in a substantial distinction between our semantics and that of Boulton [19], described in Sections 3.3.6 and 4.2.4.

[5] *const* and *chooser* are called *value* and *choosers* respectively in [151].

3.3.1 Some Definitions

We define a number of types and functions used in the static and dynamic semantics of picoELLA.

Type Name	Defined As	Typical Element
Env	$Name \rightarrow Const$	Γ
Type	$Tname + (Type \times Type)$	τ
TEnv	$Name \rightarrow Type$	T
AEnv	$(Tname + Cname) \rightarrow Type$	S

We will also use primed and subscripted versions of the typical elements. Elements of *Env* are value environments, mapping names to constants. *Type* is a type whose elements are type names or binary tuples of types. Elements of *TEnv* are type environments, which may be thought of as mapping variable *name*s to the type of their defining expressions. Elements of *AEnv* are type alias environments, which may be thought of as mapping *tname*s to their definition, or *cname*s to the type they belong to. An enumerated type is mapped to itself, and tuple types are mapped to their fully expanded definitions. This ensures monogenicity of the static semantics. A similar approach is taken in the dynamic semantics, where bottom values of a tuple type are mapped to the equivalent tuple of bottom values of the constituent types. We use *Dom* to extract the domain from a *TEnv* or *AEnv*. Environment lookup is defined as follows. $E\{(x,y)\}(z) = y$ if $z = x$ and $E(z)$ otherwise. E must be of type *Env*, *TEnv* or *AEnv*.

match is the formalisation of the matching process described informally on page 64. *match* only operates on well-typed choosers and constants of the same type. Well-typedness is defined below, as part of the static semantics. The type of *match* is $match : chooser \rightarrow const \rightarrow bool3$, where *bool3* is Kleene's ternary logic [110]. We denote truth by tt, falsity by *ff* and don't know or undefined by uu. $c = c'$ is interpreted as structural equality of c and c' in the first two clauses of *match*, and in semantic rules 3.10 and 3.29.

$$\begin{aligned}
match\ cname\ cname' &= \mathit{ff} && cname \neq cname' \\
match\ cname\ cname' &= tt && cname = cname' \\
match\ cname\ ?tname &= uu && \\
match\ tname\ c &= tt && \\
match\ (ch \mid ch')\ c &= match\ ch\ c \vee match\ ch'\ c && \\
match\ (ch, ch')\ (c, c') &= match\ ch\ c \wedge match\ ch'\ c' &&
\end{aligned}$$

match is used in rule 3.34 of the dynamic semantics.

The function $\downarrow$ is used in rules 3.10, 3.28, and 3.34 to project a constant value to the bottom value of the same type.

$$\begin{aligned}
\downarrow cname &= ?tname && cname \text{ has type } tname \\
\downarrow ?tname &= ?tname && \\
\downarrow (c, c') &= (\downarrow c, \downarrow c') &&
\end{aligned}$$

3.3.2 A Static Semantics

The static semantics of declarations is somewhat complicated by the fact that we want to ensure that programs are uniquely typable. A tuple type will therefore be expanded to its most basic constituent enumerated subtypes. In general though, the static semantics 'does the obvious thing.'

The $\_ \vdash \_ : \_$ notation is overloaded, and has the following types:

	Type	*Example*
1	$AEnv \times Tenv \times Expr \times Type$	$S, T \vdash e : \tau$
2	$AEnv \times Expr \times Type$	$S \vdash e : \tau$
3	$AEnv \times Expr \times AEnv$	$S \vdash e : S$
4	$Type \times AEnv \times Expr \times AEnv$	$\tau, S \vdash e : S$
5	$AEnv \times Tenv \times Expr \times TEnv$	$S, T \vdash e : T$

1 is used to type expressions and programs, 2 is used to derive the type of a type expression, chooser or constant, 3 returns a new type alias environment after a type declaration, 4 is used to give constructors a type τ, and 5 returns the type environment of local declarations.

Declarations

$$\frac{S \vdash \ tdecl: \ S' \qquad S', T \vdash \ program: \ \tau}{S, T \vdash \ \texttt{TYPE}\ tdecl\ \texttt{IN}\ program: \ \tau} \tag{3.1}$$

$$\frac{S \vdash \ ttype: \ \tau \qquad S, T\{(name, \tau)\} \vdash \ expr: \ \tau'}{S, T \vdash \ \texttt{INPUT}\ name: ttype\ \texttt{IN}\ expr: \ \tau'} \quad name \notin Dom(T) \tag{3.2}$$

$$\frac{}{S \vdash \ tname: \ S(tname)} \quad tname \in Dom(S) \tag{3.3}$$

$$\frac{S \vdash \ ttype: \ \tau \qquad S \vdash \ ttype': \ \tau'}{S \vdash \ ttype * ttype': \ \tau \times \tau'} \tag{3.4}$$

$$\frac{tname, S \vdash \ btype: \ S'}{S \vdash \ tname = btype: \ S'\{(tname, tname)\}} \quad tname \notin Dom(S') \tag{3.5}$$

$$\frac{}{\tau, S \vdash \ cname: \ S\{(cname, \tau)\}} \quad cname \notin Dom(S) \tag{3.6}$$

$$\frac{\tau, S \vdash \ btype: \ S' \qquad \tau, S' \vdash \ btype': \ S''}{\tau, S \vdash \ btype \,|\, btype': \ S''} \tag{3.7}$$

$$\frac{S \vdash \ ttype: \ \tau}{S \vdash \ tname = ttype: \ S\{(tname, \tau)\}} \quad tname \notin Dom(S) \tag{3.8}$$

$$\frac{S, T \vdash \ expr: \ \tau}{S, T \vdash \ name = expr: \ T\{(name, \tau)\}} \tag{3.9}$$

$$\frac{S \vdash \ const: \ \tau \qquad S, T\{(name, \tau)\} \vdash \ expr: \ \tau}{S, T \vdash \ \texttt{INIT}\ const\ \texttt{REC}\ name = expr: \ T\{(name, \tau)\}} \quad \downarrow const = const \tag{3.10}$$

Expressions

$$\frac{}{S, T \vdash \ name: \ T(name)} \quad name \in Dom(T) \tag{3.11}$$

$$\frac{S, T \vdash \ expr: \ \tau_1 \times \tau_2}{S, T \vdash \ expr[i]: \ \tau_i} \quad i = 1, 2 \tag{3.12}$$

$$\frac{S, T \vdash \ expr: \ \tau \qquad S, T \vdash \ expr': \ \tau'}{S, T \vdash \ (expr, expr'): \ \tau \times \tau'} \tag{3.13}$$

$$\frac{S \vdash \ const: \ \tau \qquad S, T \vdash \ expr: \ \tau}{S, T \vdash \ \texttt{DELAY}\ (const, expr): \ \tau} \tag{3.14}$$

$$\frac{S, T \vdash \; expr : \; \tau \qquad S \vdash \; chooser : \; \tau \qquad S, T \vdash \; expr' : \; \tau' \qquad S, T \vdash \; expr'' : \; \tau'}{S, T \vdash \; \texttt{IF}\; expr \;\texttt{MATCHES}\; chooser \;\texttt{THEN}\; expr' \;\texttt{ELSE}\; expr'' : \; \tau'} \tag{3.15}$$

$$\frac{S, T \vdash \; decl : \; T' \qquad S, T' \vdash \; expr : \; \tau}{S, T \vdash \; \texttt{LET}\; decl \;\texttt{IN}\; expr : \; \tau} \tag{3.16}$$

$$\frac{}{S \vdash ?tname : \; S(tname)} \qquad tname \in Dom(S) \tag{3.17}$$

$$\frac{S \vdash \; const : \; \tau \qquad S \vdash \; const' : \; \tau'}{S \vdash (const, const') : \; \tau \times \tau'} \tag{3.18}$$

$$\frac{}{S \vdash \; cname : S(cname)} \qquad cname \in Dom(S) \tag{3.19}$$

$$\frac{}{S \vdash \; tname : \; S(tname)} \qquad tname \in Dom(S) \tag{3.20}$$

$$\frac{S \vdash \; chooser : \; \tau \qquad S \vdash \; chooser' : \; \tau}{S \vdash \; chooser \,|\, chooser' : \; \tau} \tag{3.21}$$

$$\frac{S \vdash \; chooser : \; \tau \qquad S \vdash \; chooser' : \; \tau'}{S \vdash \; (chooser, chooser') : \; \tau \times \tau'} \tag{3.22}$$

Comments

The static semantics uses a type alias environment S, and a type environment T. Normally both would be empty initially, but this is not enforced. S contains the type of every constructor *cname*, and the type associated with a *tname*. *tname*s cannot hide *cname*s and *vice versa* (3.5, 3.6.) If this were allowed we could not type choosers unambiguously; we would not be able to choose between rule 3.19 and 3.20. For similar reasons *cname*s and *tname*s may not be redefined (3.6, 3.8.) Lambda abstracted variables *name*s, *i.e.* in `LET` constructs, may be the same as both *cname*s and *tname*s. This is safe because *name*s and *cname*s appear in different type environments. Note that typing the recursive `LET` is trivial due to the presence of the initial value *const*. In some cases, it would be possible to derive an infinite number of distinct types if this constant was absent. An example of such a circuit is `LET REC` x `=` x `IN` x. The side condition requires that the initial approximation must be equal to the bottom value of its type.

3.3.3 A Dynamic Semantics

The dynamic semantics follows the same principle as the simulator model for ELLA, described previously. The type of the semantics is $(Const\ stream \times Env \times Expr) \times (Const\ stream \times Expr)$. An input stream, initial value environment and circuit are evaluated to an output stream and a new circuit description. Within one time step the type of the dynamic semantics is $(Env \times Expr) \times\ (Const \times Expr)$. We discussed the type of the semantics previously on page 31. The fixed point computation was first described informally by Elliot [52].

The $\_ \vdash \_ \Rightarrow \_$ notation is overloaded, and has the following types:

	Type	*Example*
1	$Const\ stream \times Env \times Expr \times Const\ stream \times Expr$	$s, \Gamma \vdash\ e \Rightarrow\ s, e$
2	$Const \times Env \times Expr \times Const \times Expr$	$c, \Gamma \vdash\ e \Rightarrow\ c, e$
3	$Env \times Expr \times Env \times Expr$	$\Gamma \vdash\ e \Rightarrow\ \Gamma, e$
4	$Env \times Expr \times Const \times Expr$	$\Gamma \vdash\ e \Rightarrow\ c, e$

1 deals with input streams and whole programs, 2 passes the input at the current time step into the program, 3 computes the value environment of a local declaration, and 4 evaluates an expression.

Programs

$$\frac{}{nil,\ \Gamma \vdash\ program \Rightarrow\ nil, program} \tag{3.23}$$

$$\frac{c,\ \Gamma \vdash\ program \Rightarrow\ c',\ program' \qquad t,\ \Gamma \vdash\ program' \Rightarrow\ t',\ program''}{c :: t,\ \Gamma \vdash\ program \Rightarrow\ c' :: t',\ program''} \tag{3.24}$$

Declarations

$$\frac{c,\ \Gamma \vdash\ program \Rightarrow\ c',\ program'}{c,\ \Gamma \vdash\ \texttt{TYPE}\ tdecl\ \texttt{IN}\ program \Rightarrow\ c',\ \texttt{TYPE}\ tdecl\ \texttt{IN}\ program'} \tag{3.25}$$

$$\frac{\Gamma\{(name, c)\} \vdash\ expr \Rightarrow\ c',\ expr'}{c,\ \Gamma \vdash\ \texttt{INPUT}\ name : ttype\ \texttt{IN}\ expr \Rightarrow\ c',\ \texttt{INPUT}\ name : ttype\ \texttt{IN}\ expr'} \tag{3.26}$$

$$\frac{\Gamma \vdash\ expr \Rightarrow\ c,\ expr'}{\Gamma \vdash\ name = expr \Rightarrow\ \Gamma\{(name, c)\},\ name = expr'} \tag{3.27}$$

$$\frac{\Gamma\{(name, c)\} \vdash \ expr \Rightarrow \ c,\ expr'}{\begin{array}{l}\Gamma \vdash \ \texttt{INIT}\ c\ \texttt{REC}\ name = expr \Rightarrow \\ \qquad \Gamma\{(name, c)\},\ \texttt{INIT}\ \downarrow c\ \texttt{REC}\ name = expr'\end{array}} \tag{3.28}$$

$$\frac{\begin{array}{l}\Gamma\{(name, c)\} \vdash \ expr \Rightarrow \ c', expr' \\ \Gamma \vdash \ \texttt{INIT}\ c'\ \texttt{REC}\ name = expr \Rightarrow \ \Gamma',\ expr''\end{array}}{\Gamma \vdash \ \texttt{INIT}\ c\ \texttt{REC}\ name = expr \Rightarrow \ \Gamma',\ expr''} \quad c \neq c' \tag{3.29}$$

Expressions

$$\frac{}{\Gamma \vdash \ name \Rightarrow \ \Gamma(name),\ name} \tag{3.30}$$

$$\frac{\Gamma \vdash \ expr \Rightarrow \ (c_1, c_2),\ expr'}{\Gamma \vdash \ expr[i] \Rightarrow \ c_i,\ expr'[i]} \qquad i = 1, 2 \tag{3.31}$$

$$\frac{\Gamma \vdash \ expr_1 \Rightarrow \ c_1,\ expr_1' \qquad \Gamma \vdash \ expr_2 \Rightarrow \ c_2,\ expr_2'}{\Gamma \vdash \ (expr_1, expr_2) \Rightarrow \ (c_1, c_2),\ (expr_1', expr_2')} \tag{3.32}$$

$$\frac{\Gamma \vdash \ expr \Rightarrow \ c',\ expr'}{\Gamma \vdash \ \texttt{DELAY}\,(c,\ expr) \Rightarrow \ c,\ \texttt{DELAY}\,(c',\ expr')} \tag{3.33}$$

$$\frac{\begin{array}{l}\Gamma \vdash \ expr_0 \Rightarrow \ c_0,\ expr_0' \\ \Gamma \vdash \ expr_1 \Rightarrow \ c_1,\ expr_1' \\ \Gamma \vdash \ expr_2 \Rightarrow \ c_2,\ expr_2'\end{array}}{\begin{array}{l}\Gamma \vdash \ \texttt{IF}\ expr_0\ \texttt{MATCHES}\ chooser\ \texttt{THEN}\ expr_1\ \texttt{ELSE}\ expr_2 \Rightarrow \\ \qquad c,\ \texttt{IF}\ expr_0'\ \texttt{MATCHES}\ chooser\ \texttt{THEN}\ expr_1'\ \texttt{ELSE}\ expr_2'\end{array}} \tag{3.34}$$

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

$$c \equiv \begin{cases} c_1 & match\ chooser\ c_0 = tt \\ c_2 & match\ chooser\ c_0 = ff \\ \downarrow c_1 & match\ chooser\ c_0 = uu \end{cases}$$

$$\frac{\Gamma \vdash \ decl \Rightarrow \ \Gamma',\ decl' \qquad \Gamma' \vdash \ expr \Rightarrow \ c,\ expr'}{\Gamma \vdash \ \texttt{LET}\ decl\ \texttt{IN}\ expr \Rightarrow \ c,\ \texttt{LET}\ decl'\ \texttt{IN}\ expr'} \tag{3.35}$$

$$\frac{}{\Gamma \vdash \ cname \Rightarrow \ cname,\ cname} \tag{3.36}$$

$$\frac{\Gamma \vdash \ c_1 \Rightarrow \ c_1,\ c_1' \qquad \Gamma \vdash \ c_2 \Rightarrow \ c_2,\ c_2'}{\Gamma \vdash \ (c_1, c_2) \Rightarrow \ (c_1, c_2),\ (c_1', c_2')} \tag{3.37}$$

$$\frac{}{\Gamma \vdash ?tname \Rightarrow ?tname,\ ?tname} \quad \text{type of } tname \text{ is a } btype \tag{3.38}$$

$$\frac{\Gamma \vdash ?tname_1 \Rightarrow\ c_1,\ c_1' \qquad \Gamma \vdash ?tname_2 \Rightarrow\ c_2,\ c_2'}{\Gamma \vdash ?tname \Rightarrow\ (c_1, c_2),\ ?tname} \quad \begin{array}{l}\text{type of } tname \text{ is} \\ tname_1 \times tname_2\end{array} \tag{3.39}$$

Comments

The dynamic semantics consists of three types of rules; (i) rule 3.25 which skips type definitions which are not used in the dynamic semantics, (ii) those dealing with time, (iii) and those evaluating the circuit during each clock cycle. Rules 3.23 and 3.24 comprise the second class, the rules 3.26 to 3.39 the third class. Rule 3.24 takes a value off the input sequence. After the circuit has been evaluated using the third class of rules, the remainder of the input stream is processed. This constitutes the advance of time, with rules 3.26 to 3.39 applying only within single clock cycles. The first four rules (3.23 to 3.26) take successive constants off the input sequence and update the initial environment. The initial environment would normally be empty, but this is not enforced. The `IF` statement (rule 3.34) is strict in the sense that it always evaluates both branches. This is necessary because the new circuit description needs the new description of both branches in all cases. The two rules dealing with recursion, 3.28 and 3.29, are the most interesting. 3.28 indicates that a fixed point has been reached, and returns this value. The side condition of 3.29 ensures that as long as a fixed point has not been reached, the defining expression is re-evaluated. Note that for the re-evaluation the original expression *expr* is used, rather than the result expression of the first approximation *expr*′. The least fixed point iteration encoded by these two rules takes place within a single clock cycle, which is why we must use the initial circuit. This manifests itself when dealing with delays. Consider the following circuit.

```
LET INIT c REC x = DELAY (s, x) IN x
```

This is a delayed feedback loop, with no other components in the loop. The current state of the latch is s, and the current approximation on the wire c. c may be any value of the correct type. If $c \neq s$ then the declarative part of the `LET REC` will have the following derivation.

$$\dfrac{(*) \qquad \dfrac{\dfrac{\Gamma\{(x,s)\} \vdash\ x \Rightarrow\ s, x}{\Gamma\{(x,s)\} \vdash\ \texttt{DELAY}(s,x) \Rightarrow\ s, \texttt{DELAY}(s,x)}}{\Gamma \vdash\ \texttt{INIT}\ s\ \texttt{REC}\ x = \texttt{DELAY}(s,x) \Rightarrow\ \Gamma\{(x,s)\},\ \texttt{INIT}\ \downarrow s\ \texttt{REC}\ x = \texttt{DELAY}(s,x)}}{\Gamma \vdash\ \texttt{INIT}\ c\ \texttt{REC}\ x = \texttt{DELAY}(s,x) \Rightarrow\ \Gamma\{(x,s)\},\ \texttt{INIT}\ \downarrow s\ \texttt{REC}\ x = \texttt{DELAY}(s,x)}\ c \neq s$$

Where $(*)$ is

$$\dfrac{\Gamma\{(x,c)\} \vdash\ x \Rightarrow\ c, x}{\Gamma\{(x,c)\} \vdash\ \texttt{DELAY}(s,x) \Rightarrow\ s, \texttt{DELAY}(c,x)}$$

The result of the semantics is the same feedback loop, but with s instead of c on the output. That is, the state of the delay s is the output. This makes sense because we would not expect c to be able to influence the output of the latch, because this would correspond to a write-through during the same clock cycle of the delay. If we used the result expression of the first approximation, we would obtain the following derivation:

$$\dfrac{(*) \qquad \dfrac{(**) \qquad \dfrac{\cdots}{\Gamma \vdash\ \texttt{INIT}\ c\ \texttt{REC}\ x = \texttt{DELAY}(s,x) \Rightarrow\ \cdots}\ c \neq s}{\Gamma \vdash\ \texttt{INIT}\ s\ \texttt{REC}\ x = \texttt{DELAY}(c,x) \Rightarrow\ \cdots}\ s \neq c}{\Gamma \vdash\ \texttt{INIT}\ c\ \texttt{REC}\ x = \texttt{DELAY}(s,x) \Rightarrow\ \cdots}\ c \neq s$$

Where $(*)$ remains unchanged and $(**)$ is the following derivation:

$$\dfrac{\Gamma\{(x,s)\} \vdash\ x \Rightarrow\ s, x}{\Gamma\{(x,s)\} \vdash\ \texttt{DELAY}(c,x) \Rightarrow\ c, \texttt{DELAY}(s,x)}$$

In this case the input of the delay c has changed the output of the delay within the same clock cycle, and there is no fixed point. This is incorrect.

Instead of storing a constant c in the delay, it is also possible to store the input circuit to the delay there.

$$\dfrac{\Gamma \vdash\ \mathit{expr} \Rightarrow\ s,\ \mathit{expr}'}{\Gamma \vdash\ \texttt{DELAY}\ \mathit{expr} \Rightarrow\ c,\ \texttt{DELAY}\ \mathit{expr}'}$$

This rule is in fact incorrect because in the premise we should use the environment from the *previous time step*. If we take this approach, we have to save the previous environments, and use these in the evaluation of delays. This is the approach taken by Davies [47], and by Barringer *et al.* for their Logic+Delay language [5]. A finite upper bound on the number of previous environments which has to be saved is given in [5]. Our solution removes the need to remember the previous state because the evaluated circuit, *i.e.* its output $(s, expr)$, instead of the unevaluated circuit is saved in the delay.

An alternative encoding of the fixed point computation is given below.

$$\frac{\Gamma\{(name, c)\} \vdash \; expr \Rightarrow \; c, \; expr'}{\Gamma \vdash \; \texttt{INIT}\; c\; \texttt{REC}\; name = expr \Rightarrow \; \Gamma\{(name, c)\}, \; \texttt{INIT}\; c\; \texttt{REC}\; name = expr'} \tag{3.40}$$

$$\frac{\begin{array}{l}\Gamma\{(name, c)\} \vdash \; expr \Rightarrow \; c', expr' \\ \Gamma \vdash \; \texttt{INIT}\; c'\; \texttt{REC}\; name = expr \Rightarrow \; \Gamma', \; \texttt{INIT}\; c''\; \texttt{REC}\; name = expr''\end{array}}{\Gamma \vdash \; \texttt{INIT}\; c\; \texttt{REC}\; name = expr \Rightarrow \; \Gamma', \; \texttt{INIT}\; c\; \texttt{REC}\; name = expr''} \quad c \neq c' \tag{3.41}$$

The difference with rules 3.28 and 3.29 is that we do not use the $\downarrow$ function, but change the initial approximation in the result expression as we come out of the recursion.

3.3.4 Results About the Semantics

We have not proved any formal results about the operational semantics. We sketched a number of results such as the totality and monogenicity of the static and dynamic semantics. The dynamic semantics is monotone too. It was decided that it made more sense to prove these results in an automated version of the semantics. These results have indeed been proved in the embedding of picoELLA in LAMBDA, as we will describe in Chapter 4.

3.3.5 Alternative Semantics

With a small language such as picoELLA there are some features one could add. Associated types are the most interesting feature. Functions could be added, but would fundamentally change the flavour of picoELLA.

We will refer to the semantics presented above as the *standard semantics* in the remainder of this section. One particular question which may be asked about picoELLA's dynamic semantics is how optimistic or pessimistic its answers are. It is obvious that a bottom semantics which always delivers the bottom value of the appropriate type is the least semantics. (Assuming an element-wise ordering on semantics.) It is not at all obvious if there exists a *maximal semantics*, and what it would look like.

One simple change in the current semantics produces what we call the *greatest lower bound semantics*. If, instead of delivering the bottom value for a *uu* match, we output the greatest lower bound of c_1 and c_2 we obviously produce a more defined semantics.[6]

$$c \equiv \begin{cases} c_1 & match\ chooser\ c_0 = tt \\ c_2 & match\ chooser\ c_0 = ff \\ c_1 \sqcap c_2 & match\ chooser\ c_0 = uu \end{cases}$$

Whether this change has a corresponding hardware intuition is not clear. The relationship between the standard and greatest lower bound semantics has been formally proved in Section 4.3.3.

The standard semantics is pessimistic in its treatment of bottom values. All bottom values are considered to be distinct. In cases such as the following XOR gate

```
IF LET x = ?bool IN (x,x) MATCHES (true,true)|(false,false)
THEN false ELSE true
```

do we allow the inference that `(?bool,?bool)` going into the matching process is really either `(true,true)` or `(false,false)`? After all x is either `true` or `false`, and evaluates to the same value in both parts of the tuple. Morison, one of ELLA's designers at RSRE Malvern, expressed a willingness to apply this sort of optimisation [135]. A possible intuition for undefined values had been to regard them as sets of possible values [135]. In that case the XOR gate above would indeed output `false` rather than the pessimistic `?bool`. It is straightforward to formalise

[6] This variation was suggested by Simon Finn of Abstract Hardware Ltd.

a basic version of this semantics. For example, the result of a tuple expression would be the product set of the two sets obtained from the subderivations.

$$\frac{\Gamma \vdash \; expr_1 \Rightarrow \; c_1, \; expr_1' \qquad \Gamma \vdash \; expr_2 \Rightarrow \; c_2, \; expr_2'}{\Gamma \vdash \; (expr_1, expr_2) \Rightarrow \; c_1 \times c_2, \; (expr_1', expr_2')}$$

Where $c \times c'$ is defined as $\{(x, y) | x \in c, y \in c'\}$. The matching process would be reduced to set theoretic intersections and subsets. Choosers would be converted to sets as follows.

$$\begin{array}{rcl}
cname & \rightarrow & \{cname\} \\
tname & \rightarrow & \{c | c \text{ of type } tname\} \\
ch \,\mathtt{|}\, ch' & \rightarrow & ch \cup ch' \\
(ch, ch') & \rightarrow & ch \times ch'
\end{array}$$

The complement $\overline{ch}$ of a chooser ch can be defined to be all the constants of the same type which are not in the set ch. We choose the `THEN` branch of an `IF` if the input c is a subset of the chooser, and the `ELSE` branch if it is a subset of the complement. If c contains elements of both ch and $\overline{ch}$ we cannot decide between the two branches. The output of the `IF` becomes the union of the outputs of the two branches in this case.

$$c \equiv \begin{cases} c_1 & \text{if } c \subseteq ch \\ c_2 & \text{if } c \subseteq \overline{ch} \\ c_1 \cup c_2 & \text{if } c \cap ch \neq \{\} \wedge \; c \cap \overline{ch} \neq \{\} \end{cases}$$

Recursion is handled without problems in this approach also. While this increases our outputs, this *set semantics* is not optimal. Consider the following circuit.

```
LET x = IF ?bool THEN true ELSE false IN
LET y = IF x THEN false ELSE true IN
IF (x,y) MATCHES (true,true)|(false,false) THEN false ELSE true
```

We invert an unknown signal x, later we match the two. Intuitively y should be x's complement, but we do not have this information. Thus the output from the tuple

(x, y) is not {`(true,true)`, `(false,false)`} but also includes {`(true,false)`, `(false,true)`}. The final result is therefore {`true, false`} instead of the desired {`true`}. The remedy for this problem would be to separately derive an output for every member of a set, but this causes problems in the computation of fixed points. In the set semantics any undefined value matches with the chooser which contains all constructors of a type, *e.g.* `true|false`. In ELLA and picoELLA, this would result in neither a match nor a no-match.

The previous semantics have used information within one time step only. We can try to recover some of ELLA version 3.0's concepts; *i.e.* to use the output from the previous time step as the initial approximation for recursive `LET` statements. This may be desirable from a hardware point of view, where a state may be retained by a wire for some time. Thus, rather than starting with the bottom approximation at the start of every fixed point iteration we use the output from the previous time step. In some cases the fixed point iteration would oscillate. By taking a fixed upper bound on the number of iterations we can still guarantee termination. It is not clear if we retain the monotonicity of the semantics (assuming that the very first circuit description has all bottom initial values.) This method can give rise to fixed points which are not least fixed points. We can sanitize this crude method by detecting the fact that no fixed point will be reached (the evaluation loops with more than one distinct element in the loop.) Two options are open to us; we can output undefined at this point, reflecting the fact that no fixed point was reached, or we can compute the least fixed point, using the standard semantics. In the former case neither this nor the standard semantics is always more defined than the other. In the latter case the ordering relationship is not clear. Clearly, in some cases this semantics will be more defined than the standard semantics. It would seem that this semantics will never be less defined than the standard semantics, but this is only a conjecture at present. In fact, the method of detecting a loop gives us a family of semantics, depending on after how many steps we suspect that the circuit is oscillating.

We may combine various features of the above semantics in order to try to obtain the maximal semantics. There are some interesting avenues to be explored using these semantics.

3.3.6 Different Approaches to ELLA Semantics

A number of groups in the UK have reported on semantics for subsets of ELLA.

Davies' language $\mathcal{L}$ [47] is a small language into which ELLA may be translated for the purposes of equivalence checking. The constructs of $\mathcal{L}$ are: constants S_S, variables V_S, delays $\Delta_k\alpha$, pairing (α, β), case clause $\Box\alpha : \mathcal{A}$, and recursion $\mu v.\alpha$. It is a subset of picoELLA; it lacks indexing and non-recursive `LET` constructs. A more important restriction is that bottom values are omitted, case statements must be total, and that feedback loops must be delayed.[7] These restrictions are enforced by a static semantics. The dynamic semantics has type:

$$evaluate : time \rightarrow valuation \rightarrow value$$

A valuation gives a value for a *name* at every time: $valuation = name \rightarrow time \rightarrow value$. The clauses for recursion $\mu v.\alpha$ and the delay $\Delta_k\alpha$ are the most interesting. The expression for a delay is simply evaluated at the previous time step; at time zero k is returned. In the case of a recursive feedback all occurrences of the variable v must be protected by a delay. All previous environments are generated, and the subexpression α is evaluated.

$$\begin{array}{rcl}
evaluate_{t,V}v & = & V\ v_t \\
evaluate_{0,V}(\Delta_k\alpha) & = & k \\
evaluate_{t+1,V}(\Delta_k\alpha) & = & evaluate_{t,V}(\Delta_k\alpha) \\
evaluate_{t,V}(\mu v.\alpha) & = & evaluate_{t,V'}(\mu v.\alpha) \\
\text{where } V' & = & V \cup (\cup_{i=1}^{t-1}\{(v_{t-i}, evaluate_{t-i,V}\alpha)\})
\end{array}$$

Note the subtle interplay between the delay and recursion: a variable can occur free inside a delay only if the delay is part of a recursive `LET`. As a result the

[7] This means that we cannot get bottom value results.

value of the variable is defined for all previous times in the environment V'. The number of previous environments to be computed for each recursive `LET` rises exponentially with time in this semantics. Equivalence of expressions is defined as identical output behaviour over all time. Davies notes that a finite program with finite data types can exhibit only a finite behaviour given a *constant* input. After this time, which is related to the number of delays and the size of the data type of variables in recursive `LET` statements, the behaviour repeats itself. It is shown that program equivalence is the same as comparing both programs on every possible input combination for this length of time.

The Logic+Delay language of Barringer *et al.* [5] is a subset of Davies' language $\mathcal{L}$. In Logic+Delay a circuit is not a single expression, but a set of bindings to variables. Bindings may be mutually recursive, and are evaluated simultaneously until a fixed point is found. If the circuit oscillates all bindings become undefined.

$$\frac{\bigwedge_{i \in 1..k} h \vdash_{|h|} e_i \Rightarrow_\Delta b_i}{h \xrightarrow{\{v_i = e_i | i \in 1..k\}}_\Delta h \dagger\!\!\dagger \{v_i \mapsto b_i | i \in 1..k\}}$$

This rule states that all bindings are evaluated simultaneously, after which the environment h is updated. The following rule detects a fixed point of the circuit. The $\cdots$ notation is a shorthand for the transitive closure of the $\longrightarrow_\Delta$ relation.

$$\frac{h' = (h \frown (last\ h)) \dagger\!\!\dagger I \qquad h' \xrightarrow{cir}_\Delta \cdots \xrightarrow{cir}_\Delta h'' \qquad h'' \xrightarrow{cir}_\Delta h''}{h \overset{cir}{\underset{I}{\rightsquigarrow}} h''}$$

A similar rule is used to detect unstable loops.

$$\frac{h' = (h \frown (last\ h)) \dagger\!\!\dagger I \quad h' \xrightarrow{cir}_\Delta \cdots \xrightarrow{cir}_\Delta h'' \quad h'' \xrightarrow{cir}_\Delta h''' \xrightarrow{cir}_\Delta \cdots \xrightarrow{cir}_\Delta h'' \quad h'' \neq h'''}{h \overset{cir}{\underset{I}{\rightsquigarrow}} (h \frown (UNDEF \dagger I))}$$

UNDEF is an undefined state. The following rule uses the *negative judgement* $\not\vdash_t$ to determine when an expression evaluates to undefined. The t subscript indicates the time. This facilitates the evaluation of the unit delays, which just compute the value of their input expression at the previous time step. Of course, this means that a number of previous environments h need to be kept, as discussed in

Section 3.3.3.

$$\frac{h \not\vdash_t rhs \Rightarrow_\Delta 0 \qquad h \not\vdash_t rhs \Rightarrow_\Delta 1}{h \vdash_t rhs \Rightarrow_\Delta ?}$$

In our approach an iterative method gives us the fixed point solution of a circuit (rules 3.28 and 3.29.) We compare the previous approximation with the current one; if they are equal we have found a fixed point, if not we continue. In Barringer's work the stability or instability of a circuit is detected at the higher level of complete evaluations. That is, whole states are compared to find a fixed point for the whole circuit. This is closer to the approach we used for microELLA [70]. In the picoELLA and microELLA semantics we need only keep track of the last approximation (as a value and state respectively) because the semantic model guarantees that a fixed point will be reached. In Logic+Delay this is not the case, and a large (but finite) number of states may have to be kept to detect a (non-occurrence of a) fixed point.

Barringer *et al.* [4] describe an ELLA subset called boolean kernel ELLA. It is a large subset which includes functions. Program constructs are mapped onto IO automata, and are composed using a causal product. A causal product consists of the product of the substates with internal communication lines removed. Internal data exchange is therefore hidden from observers. The causal product constrains transitions to those that are consistent. Temporal properties may be stated and proved using this state-based approach.[8]

microELLA [70] is the only subset to tackle sequences explicitly. We have described this work at the start of this section.

HOL ELLA [19,17] has been given a semantics by Boulton by mapping it into HOL. We have discussed this work in the context of partially formal behaviour functions in the previous chapter. The HOL ELLA subset does not contain sequences, but does deal with a substantial subset. It is the only ELLA subset to include macros. For a technical comparison of HOL ELLA and embedded picoELLA see Section 4.2.4.

[8] This work has been extended to cover Core ELLA [6], superseding [136,92], discussed below.

Recent work by Harrison at Cambridge on HOL ELLA has removed the explicit bottom values [18, Section 7.8]. This resulting semantics implicitly takes an approach similar to our *set semantics* (page 77 above.) By treating undefined values not as an extra element in the value domain, but as an implicitly defined value he obtains the same result. Using Hilbert's ε operator a bottom value is the disjunction of all possible constructors. A result of this method is that a bar chooser which includes all possible values will match with the bottom value. For example,

```
IF ?bool MATCHES true|false THEN true ELSE false
```

will output `true`. In our standard semantics it delivers `?bool`. A problem arises through the use of the ε operator. It is possible to identify *all* undefined values which describe the same set of values. This means that unrelated signals in a circuit can be unified, which has no hardware or semantic intuition.

Morison and Hill have defined a large subset of ELLA, called Core ELLA[136]. Automated tools exist to map full ELLA into Core ELLA. Macros, sequences, function types, *etc.* may be transformed to Core ELLA using this Software Transformational System, and so be given a derived semantics. Note that this set of tools is not formalised, and that it is therefore not possible to reason formally about this mapping. For pragmatic reasons Core ELLA contains some constructs which could have been translated into more primitive ELLA. Functions, for example, can be removed (as in picoELLA), but the resulting lack of structure makes it harder to reason about programs. As Core ELLA is intended to be used by practical tools and applications, its composition was a trade-off between minimality and usability. Core ELLA's static semantics has been defined in [136] by mapping Core ELLA to Kernel ELLA. Kernel ELLA is a set of data structures into which a static semantically correct Core ELLA program may be translated. The translation from Core to Kernel ELLA is defined in an operational semantics style. A dynamic semantics for Kernel ELLA has been defined in [92]. The semantics is an extension of Davies' semantics for his language $\mathcal{L}$. Delayless feedback loops do not seem to be addressed in this work.

3.4 Formal Semantics for Other Hardware Description Languages

Research into formal semantics for VHDL has been described earlier in this chapter (page 49.) Few hardware notations have been given a formal semantics.

Cardelli [36] gives a denotational semantics to a language to describe analogue circuits, and an operational semantics to a CCS-like language.

In Section 2.3 we already discussed work by Brock *et al.* on a formal operational semantics for an HDL which has been embedded in the Boyer-Moore theorem prover [22,23].

CIRCAL, an HDL based on process algebras, has been given a formal semantics by Moller in [132].

FUNNEL [167,168] was designed with a formal semantics in mind. A model theoretic semantics based on sheaves and initial algebras is provided. An operational semantics, defined by term rewriting, is given via a translation from FUNNEL into OBJ3 [68].

Johnson's HDL DAISY [103] has a formal semantics based on Scott-Strachey denotational semantics [169]. DAISY is a small lazy functional language in which time is represented by the use of streams. A 'hardware description with recursion equations' notation based on DAISY has been used by O'Donnell to provide more than one semantics for circuit descriptions [140]. These include shift register simulations, net-list extraction, and lay-out generation semantics.

A stream-based functional notation with a formal semantics has also been used by Delgado Kloos to describe hardware [49].

Other related research includes Rossen's work on Ruby [155,156].

Chapter 4

Embedding picoELLA in Lambda

This chapter contains an introduction to the LAMBDA proof system [64], the definitions used in the embedding of picoELLA in LAMBDA, and a detailed description of the main result of the embedding.

4.1 The Lambda Proof Assistant

Where we have mentioned LAMBDA until now, we have not distinguished the logic and its implementation. Here we will first describe the logic which the LAMBDA proof system uses, and then how it is used in practice. As LAMBDA's logic is different from the more commonly used classical logics, we also describe these differences explicitly.

4.1.1 Lambda's Logic

In our work we use LAMBDA version 3.2 [57], which has the same constructive logic as all earlier versions. LAMBDA version 4.0 and later versions use a different logic [58], which is also used by the HOL system [79]. Henceforth when we omit the version of the LAMBDA system, we will assume that version 3.2 is used. LAMBDA's logic is a higher-order constructive polymorphic logic of partial terms [57,160]. In the remainder of this section we will explain different aspects of the logic.

Logical inference rules contain a conclusion and zero or more premises. Each of these is written as a sequent, and has a conclusion on the right hand side of the turnstile, and two hypothesis lists, known as the G and H lists. The G list usually contains existence hypotheses (explained below), and the H list the remaining ('conventional') hypotheses. As an example, consider the following raw LAMBDA output.

```
2:  E r1' $ E r' $ G //
leaves r1' == nodes r1' + 1 $ leaves r' == nodes r' + 1 $ H
|- leaves (Node (r1',r')) == nodes (Node (r1',r')) + 1
1:  G // H |- leaves Leaf == nodes Leaf + 1
--------------------
G // H |- (forall t.  leaves t == nodes t + 1)
```

This rule has two premises, the first of which has no hypotheses. The dollar signs separate individual hypotheses, and the `G` and `H` indicate the ends of the G and H hypothesis lists respectively. The slashes `//` separate the G and H lists. We will pretty-print this as

```
***** Premise 2 *****
1:  E r1'
2:  E r'
1:  leaves r1' == nodes r1' + 1
2:  leaves r' == nodes r' + 1
⊢ leaves (Node (r1',r')) == nodes (Node (r1',r')) + 1
***** Premise 1 *****
⊢ leaves Leaf == nodes Leaf + 1
--------------------
⊢ ∀t.  leaves t == nodes t + 1
```

We use $\vdash$, $\forall$, $\exists$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ instead of LAMBDA syntax `|-`, `forall`, `exists`, `/\`, `\/`, `->`, and `<->` respectively. All object-level variables are printed in *italics*, and meta-variables constructors and functions in `typewriter` font. Note that the

G and H lists are numbered separately, with the G list displayed first. Sometimes we will omit the numbering of the premises and hypothesis. When listing theorems we will often omit the dashed line. A more extensive overview of our notation is given in Appendix A.

The propositional logic fragment of LAMBDA is standard, and contains rules such as `andR`:

```
***** Premise 2 *****
⊢ Q
***** Premise 1 *****
⊢ P
--------------------
⊢ P ∧ Q
```

The law of the excluded middle, and the strong axiom of choice are not part of the logic, because it is constructive. Thus case analysis on truth values is not possible, and double negations cannot be eliminated. There are some additional functions to manipulate the hypothesis lists: `permh:` *int list* `->` *unit* permutes the H list hypotheses according to the list of integers. The function `permg` is similar. `htog:` *int* `->` *unit* and `gtoh` can be used to move hypotheses from the H to G list and *vice versa*. These operations are also available as tactics *e.g.* `permhTac`, `htogTac`. The G and H lists behave as multisets.

Lambda's Type System

LAMBDA's type system can be divided into object-level types, and meta-types [65, Section 2.4]. Object-level types are the same as ML's type system [90] without references, and with some added restrictions on function types in abstract data types. Object-level types are composed of data types, record types, tuple types and function types. Object-level types may be polymorphic. Examples are `nil:` *'a list*, and `UNDEFINED:` *'a*. An example of an object-level function is the following

```
fun sub 0 = 0 | sub (S n) = n;
```

It has type `sub`: *natural* `->` *natural*. An equivalent object-level lambda term is

```
fn 0 => 0 | S n => n
```

Higher-order object-level function types are allowed. Function composition could be defined as follows, for example.

```
infix o; fun f o g = fn x => f (g x);
```

The type of `o` is `o`: ('a `->` 'b) `*` ('c `->` 'a) `->` 'c `->` 'b. We will see more examples of functions later. All object-level types, which do not contain a function type are equality types. Equality types such as `nil`: *natural list* may be compared using the equality function `=`, which is provided as part of the LAMBDA libraries. Equality types are indicated by a double prime, *e.g.* ''a `*` ''a *list*.

Meta-types are composed of the type of truth values Ω, and the meta-type function arrow. Thus the truth values `TRUE` and `FALSE` have type Ω. Meta-level functions, or syntactic functions, may be named or unnamed [64, Sections 2.16 and 2.18]. The former are called abbreviations, the latter meta-level lambda terms. `F` is an example of an abbreviation:

```
val F#(x) = x → FALSE;
```

and an unnamed meta-level function

```
lam y.  y == 1 ∨ F#(E y)
```

Unnamed syntactic functions are not directly accessible to the user, but may be generated by the system. The meta-level `lam` and `F#` correspond to the object-level `fn`, and `fun` respectively. Meta-level functions may use and return both object and meta-level values, whereas object-level functions can operate on object-level values only. Thus we could not rewrite `F` above as a `fun` or `fn`. A second difference is that object-level functions may be higher-order, but meta-level functions must be first order, *i.e.* they cannot manipulate other syntactic functions.

In the rule `andR` shown previously, `P` and `Q` are variables of type omega, because they are truth-valued terms. When proving a result, variables in the goal are normally rigid (non-flexible) but may be flexible [64, Section 2.19.2]. A rigid variable cannot be specialised; it will remain a variable. Flexible variables, on the other hand, can be instantiated with another term. A flexible variable may be regarded as standing for a particular term, but we have not decided exactly which term. Rigid variables require a proof to be schematic in the variable, and ensure that a general result rather than an instantiation of the result is proved. We will see some applications of flexible variables in Section 5.1.

We will generally omit the 'object-level' adjective, but always specify 'meta-level' explicitly.

A Logic of Partial Terms

LAMBDA implements a logic of partial terms: a term does not necessarily denote. To distinguish denoting and non-denoting terms, the existence predicate `E` is provided. For example, we can prove that $\vdash$ `E 1`. All terms built from data type constructors (which we describe below) and existing terms denote. Non-denoting terms may be created using functions, and implicit descriptions. The polymorphic undefined object `UNDEFINED` may be defined using implicit descriptions, discussed below. Functions, and partial function applications to existing terms denote, but a fully applied function need not denote. That is, functions are strict but need not be total. For example, the application of the infinitely recursing function
`fun f` x `= f` x; does not denote for any x:

$$\boxed{\vdash \forall x.\ \ \texttt{NOT (E (f } x\texttt{))}}$$

To compare object-level terms we have an equality predicate `==`, and an equivalence predicate `===`. Two terms are equivalent if, whenever either denotes, they denote the same thing. Equality may be defined in terms of equivalence as follows.

```
⊢ (x == y) ↔ (x === y ∧ E x ∧ E y)
```

The difference between equality and equivalence is that any two non-denoting terms are equivalent but not equal. Note that `==` and `===` have type `Ω * Ω -> Ω`. Boolean equality `=` has type `=: "a * "a -> bool`. *"a* denotes any equality type, *i.e.* any object-level type which does not contain any function types. The function `=` returns a boolean value, rather than a truth value as these are not the same in LAMBDA (see Section 4.1.3.) One of the axioms characterising `=` is the following rule.

```
⊢ (x = y == true) ↔ x == y
```

The quantifiers $\forall$ and $\exists$ range over existing objects only, so that $\forall x.$ `E` x always holds. This is also apparent from the rule `allR`:

```
1:  E r'
⊢ P#(r')
--------------------
⊢ ∀x.  P#(x)
```

This rule states that to prove a universally quantified proposition `P`, it is enough to prove the proposition with a free variable. The term `E` *r'* states that the variable *r'* denotes. The prime indicates that *r'* is a restricted variable; it cannot be instantiated with a term in which *r* occurred free [64, Section 2.19.3]. LAMBDA is a higher-order logic so that we can quantify over functions, relations, functions of functions, *etc.* We cannot quantify over meta-types and hence neither over truth values.

The iota operator ι is used to give implicit descriptions. The unique object which satisfies `P` is given by $\iota x.$ `P#(`x`)`. If no such object exists, or more than one such object exists the term is undefined. The undefined object `UNDEFINED`, could be defined as $\iota x.$ `FALSE`. That is, the unique x such that it satisfies the condition `FALSE`. No denoting object can satisfy this condition. In LAMBDA version 3.1

some restrictions were placed on the use of the iota operator [57]. As a result it is not available directly to the user. The rule `iotaExistsR`, available until LAMBDA version 2.1, gives a good intuition for the ι operator:

```
***** Premise 3 *****
⊢ E x
***** Premise 2 *****
1:  E x
⊢ P#(x)
***** Premise 1 *****
1:  E y'
1:  P#(y')
⊢ x === y'
--------------------
⊢ E (ιx.  P#(x))
```

To prove that an iota expression exists, we must (i) prove that a witness exists, (ii) that it satisfies `P`, and (iii) the witness is unique.

Defining New Data Types and Functions

LAMBDA provides ML-style data type and function definitions [90] to allow the user to specialise the system to his or her needs. Data types definitions are the same as in ML with the following exceptions. References cannot be used, and a data type must not occur in the domain of a function type in its own definition either directly or via mutual recursion or a type abbreviation [64, Section 2.4.3], [57]. Given a definition the system returns a number of rules axiomatising it. To illustrate this, we will describe how a binary tree may be formalised.

```
datatype tree = Leaf | Node of tree * tree;
```

Four different types of rules are returned by LAMBDA:

Existence Rules Each of the constructors denote (rules `tree'ex'Leaf`, `Node'ex'0`), and applications to existing objects also denote (`tree'ex'Node`), *i.e.* constructors which take arguments are total functions.

```
****** tree'ex'Leaf ******
--------------------
⊢ E Leaf
```

```
****** tree'ex'Node ******
--------------------
1:  E v1
2:  E v
⊢ E (Node (v1,v))
```

```
****** Node'ex'0 ******
--------------------
⊢ E Node
```

Every term which is composed only of data type constructors denotes; it is only when we use function application and implicit descriptions that we can obtain non-denoting objects.

Equality Rules Constructor applications are equal if their arguments are equal. (This results in a trivial rule for the `Leaf` constructor as it takes no arguments. The `tree'eq'Leaf` rule can be derived using the reflexivity of `==` and `tree'ex'Leaf` above.)

```
****** tree'eq'Node ******
1:  v3 == v1
2:  v2 == v
⊢ R
--------------------
1:  Node (v3,v2) == Node (v1,v)
⊢ R
```

```
****** tree'eq'Leaf ******
⊢ R
--------------------
1:  Leaf == Leaf
⊢ R
```

Inequality Rules For every combination of distinct constructors there is an inequality rule. For example:

```
***** tree'ineq'Leaf'Node *****
--------------------
1:  Leaf == Node (v1,v)
⊢ R
```

Induction Rules Every data type has an associated induction principle, which axiomatises the initial algebra semantics for the data type [57]. For non-recursive data types, this reduces to a case analysis. The rule we obtain for `tree` is `tree'ind`:

```
***** Premise 2 *****
1:  E r1'
2:  E r'
1:  Ptree#(r1')
2:  Ptree#(r')
⊢ Ptree#(Node (r1',r'))
***** Premise 1 *****
⊢ Ptree#(Leaf)
--------------------
1:  E w
⊢ Ptree#(w)
```

To prove a property `Ptree` of all (existing) trees w, we have to (1) prove it for a leaf, (2) prove it for all nodes, assuming `Ptree` holds for all subtrees *r1'*, *r'*.
The function `processFun` transforms the above rules into rewrite rules. These rewrite rules may be used with standard rewrite tactics which facilitates their use. For example, `tree'ex'Leaf` becomes

```
⊢ E Leaf === TRUE
```

Functions are defined as in ML, with the added restriction that patterns must not be overlapping.

```
fun wrong 0 = 1 | wrong n = S n;
fun right 0 = 1 | right (S n) = S (S n);
```

This leads to somewhat more verbose function definitions. As with data type definitions, LAMBDA returns a number of rules which axiomatise the behaviour of functions. These may be divided into the following categories: (1) a rewrite rule for each function clause, (2) existence rules, and (3) a minimality rule. As an example we define the functions `leaves` and `nodes` structurally on trees.

```
fun leaves Leaf = 1 | leaves (Node (l,r)) = leaves l + leaves r;
fun nodes Leaf = 0 | nodes (Node (l,r)) = S (nodes l + nodes r);
```

For `leaves` above, the rewrite rules would be the following:

```
***** leaves'eq'1 *****
--------------------
⊢ leaves Leaf === 1
```

```
***** leaves'eq'2 *****
--------------------
⊢ leaves (Node (l,r)) === when#(E l ∧ E r,leaves l + leaves r)
```

The term `when#(P,Q)` is equivalent to `Q` if `P` is `TRUE`. There is only one existence rule, namely that the function exists (`leaves'ex'0`.)

```
⊢ E leaves
```

For curried functions partial applications of the function exist, provided the arguments exist. A fully applied function need not denote because although functions are strict they may be partial. Finally the minimality rule `leaves'min` states that any function *leavesX* with the same behaviour as `leaves` is equal to `leaves`.

```
***** Premise 3 *****
⊢ E leavesX
***** Premise 2 *****
⊢ ∀l,r.  leavesX (Node (l,r)) === leavesX l + leavesX r
***** Premise 1 *****
⊢ leavesX Leaf === 1
--------------------
1:  E (leaves v)
⊢ leaves v == leavesX v
```

We can now prove general properties involving trees, `leaves` and `nodes` such as $\forall t.$ `leaves` t `==` `nodes` t `+ 1`. An example LAMBDA session proving this property is shown in the following section.

It becomes cumbersome to use the above rules individually so we created some functions which take the rules and return a rewrite tactic.

```
val pureLeavesTac = mkPureTac rules "leaves";
val leavesTac = mkTac rules "leaves";
val treeInduct = lookupRule rules "tree'ind";
```

The tactic `pureLeavesTac` only rewrites (sub)terms involving the function `leaves`. `leavesTac` uses the standard rewrite library in addition to `leaves`'s rewrite rules. `treeInduct` is another name for `tree'ind`. `treeExRules`, `treeEqRules`, `treeEqConjRules` rule lists contain the existence, equality, and inequality rules involving trees respectively. These, and some other related rules, are collected into the `treeRules` rule list. Similarly named lists and functions exist for all data types and functions we will encounter.

4.1.2 Using the Lambda System

The LAMBDA proof assistant is implemented in the functional language ML [90], which it also uses as its command language. ML was originally designed as a special purpose language to implement proof systems in a safe manner. It is not possible in LAMBDA to prove logically false results, unless axioms are introduced by the user. We never use this facility.

To prove a result in LAMBDA one sets the current goal to be proved to the desired result. This results in the trivial rule 'if the result holds, then the result holds.' For example, using the definitions of the previous section, proving that the number of nodes of a binary tree is one less than the number of leaves in a tree, results in the following initial goal.

```
***** Level 1 *****
***** Premise 1 *****
⊢ ∀t.  leaves t == nodes t + 1
--------------------
⊢ ∀t.  leaves t == nodes t + 1
```

We strip the universal quantification off the right hand side, in order to apply the tree induction rule `treeInduct`. The rule `allR` allows us to do this.

```
1:  E r'
⊢ P#(r')
--------------------
⊢ ∀x.  P#(x)
```

If we apply this rule to premise 1 the context `P` and meta-variable x of `allR`'s conclusion are unified with `leaves` t `== nodes` t `+ 1` and t respectively. From this we can conclude that the new conclusion of the goal becomes `P#(`x`)`, *i.e.* `leaves` t' `== nodes` t' `+ 1`. By default LAMBDA uses higher-order unification when it applies rules and tactics, but it is possible to use matching instead [64, Section 2.19]. Matching is faster but less general.

```
> apprl allR;


***** Level 2 *****
***** Premise 1 *****
1:  E t'
⊢ leaves t' == nodes t' + 1
--------------------
⊢ ∀t.  leaves t == nodes t + 1
```

`>` is LAMBDA's prompt, and `apprl` stands for apply rule. Previous goals are retained; after a rule or tactic application the new goal is pushed onto the goal stack. Undoing the last proof step amounts to popping the top element off the stack. It is also possible to push and pop whole proof stacks, which is useful when we want to prove a lemma during the course of a proof. We can now apply the tree structural induction rule `treeInduct` of page 92 resulting in:

```
***** Level 3 *****
***** Premise 2 *****
1:  E r1'
2:  E r'
1:  leaves r1' == nodes r1' + 1
2:  leaves r' == nodes r' + 1
⊢ leaves (Node (r1',r')) == nodes (Node (r1',r')) + 1
***** Premise 1 *****
⊢ leaves Leaf == nodes Leaf + 1
--------------------
⊢ ∀t.  leaves t == nodes t + 1
```

The last two hypotheses of the second premise are the induction hypotheses. Using the rewrite rules for `leaves` and `nodes` premise 1 is discharged easily. The following tactic application accomplishes this.

```
applyTac (leavesTac thenT nodesTac);
```

Applying this tactic to the second premise gives the following goal:

```
***** Level 5 *****
***** Premise 1 *****
1:  E r1'
2:  E r'
1:  leaves r1' == nodes r1' + 1
2:  leaves r' == nodes r' + 1
⊢ E (nodes r1') ∧ E (nodes r')
--------------------
⊢ ∀t.  leaves t == nodes t + 1
```

Assuming we have proved that `nodes` is total, we can also discharge this premise:

```
> applyTac (doRule andR thenLT [idT,permgTac[2]] thenR
#           nodesTotalR);

***** Level 6 *****
--------------------
⊢ ∀t.  leaves t == nodes t + 1
```

The tactic splits the right hand side into two subgoals, and moves *r'* to the front in the second subgoal. It then applies `nodesTotalR` to both subgoals, resulting in `E (nodes` *r1'*`)` and `E (nodes` *r'*`)` respectively. This discharges the two subgoals. We can save this result as a derived rule `lemmaT`. This is not a very useful form in which to have the result, however. We therefore also derive `lemmaR`:

```
--------------------
1:  E t
⊢ leaves t == nodes t + 1
```

We can introduce the result as a hypothesis using `lemmaL`, if we know that t exists:

```
1:  E t
1:  leaves t == nodes t + 1
⊢ P
--------------------
1:  E t
⊢ P
```

These three versions of the same result we call the 'T', 'R' and 'L' versions respectively. For certain results we also have a 'U', 'F' and 'E' version. The unfold 'U' version (`lemmaU` below) is used to replace subexpressions in the conclusion, and the fold 'F' version is the inverse rule.

```
1:  E t
⊢ P#(nodes t + 1)
--------------------
1:  E t
⊢ P#(leaves t)
```

The default is to replace all subexpressions `leaves` t anywhere in the two hypothesis lists and the conclusion. There are two ways in which selected subexpressions may be changed. The first is by backtracking through possible unifications, but this becomes cumbersome, especially if there are a large number of possible unifications. Alternatively one can guide the unification using a mouse and clicking on the desired subterms in a pop-up window. (This may also be accomplished using the textual interface.) Often the 'U' and 'F' versions of rules use meta-level lambda expressions introduced on page 87. The equality or equivalence 'E' version is a rule which may be used in the rewriting tactics. `lemmaE` is the rewrite rule version of `lemmaT`.

```
⊢ leaves t === when#(E t,nodes t + 1)
```

Goal-Directed Theorem Proving

The proof we gave above was a backward proof: we start with what we want to prove, and at each step we reduce the goal, possibly producing a number of subgoals. Each of the subgoals has to be discharged to prove the result. The conclusion of the rule does not change throughout the proof. This is a convenient way of performing proofs because it allows a natural top-down approach. A forward style of proving results is more cumbersome because a proof tree has to be constructed starting from the leaves. It is very hard to know the exact leaves and rule applications which will lead to the desired goal. It is much easier to start with the desired goal, and use a divide-and-conquer approach. However, forward application of rules can be useful to slightly change the conclusion of a result which has been proved previously.

In Section 5.2.2 we apply the goal-directed style of theorem proving to the embedded operational semantics.

We will now give a brief introduction to forward theorem proving, and how it is supported in LAMBDA. `forwardApprl` is a LAMBDA function which, given a natural number n and a rule r, unifies the nth premise of r with the conclusion of the current goal [64, Section 2.15]. `forwardMapprl` is similar but uses matching rather than unification. (In backward rule applications `apprl` r unifies the conclusion of r with the first premise of the current goal.) Consider the rule `impR` (we have condensed the output somewhat to save space):

```
P ⊢ Q
--------------------
⊢ P → Q
```

In backward theorem proving we would apply this rule when we have an implication on the right hand side of the turnstile. The new goal would be to prove that `Q` is true assuming that `P` holds. In a goal-directed proof it moves an assumption into the result. The command `forwardApprl 1 impR` thus unifies premise 1

`P ⊢ Q` with the conclusion of the current goal. In the example below `reflexH` uses the first hypothesis to prove the conclusion, `monoH` introduces a new hypothesis, `or1R` introduces a disjunction on the right hand side, and `andL` a conjunction on the left hand side.

```
(* Proof system output:  *)    (* Definition of applied rule:  *)
> pushRule libPenv reflexH;
***** Level 1 *****
--------------------           --------------------
P ⊢ P                          P ⊢ P
> forwardApprl 1 monoH;

***** Level 2 *****            ⊢ Q
--------------------           --------------------
Q, P ⊢ P                       P ⊢ Q
> forwardApprl 1 or1R;

***** Level 3 *****            ⊢ P
--------------------           --------------------
Q, P ⊢ P ∨ R                   ⊢ P ∨ Q
> forwardApprl 1 andL;

***** Level 4 *****            P, Q ⊢ R
--------------------           --------------------
Q ∧ P ⊢ P ∨ R                  P ∧ Q ⊢ R
> forwardApprl 1 impR;

***** Level 5 *****            P ⊢ Q
--------------------           --------------------
⊢ Q ∧ P → P ∨ R                ⊢ P → Q
```

A rule such as `andR` (page 86) poses a problem when using forward proof strategy because it combines two proof trees. The proof of `P` and the proof of `Q` are merged to form a proof of `P ∧ Q`. In general a rule with more than one premise combines a number of proof trees. This effect can be achieved in a pleasant manner in LAMBDA. Recall from Section 4.1.2 that a proof is a stack of rules, reflecting the evolution of the proof. We can also stack proofs. That is, to prove a lemma during a proof, we just push the lemma on the proof stack, prove it and pop it off the stack. This leaves us with the original proof. If we have two proofs on the goal stack, we can combine them into one proof stack, and thus achieve the effect of a forward rule application of, for example `andR`. In general a rule with n premises will combine the top n proofs into one proof. Mick Francis of Abstract Hardware Limited provided the basis for the implementation of the `genMergeProofTrees` function which combines proof trees. It has type `genMergeProofTrees:` *bool* `->` *int list* `->` *rule* `->` *unit* `->` *unit*. The boolean value indicates whether intermediate results are displayed. The integer list `[`i_1`,...,`i_N`]` shows which premise of the rule is unified with proof tree $1, \ldots, N$. A unit-to-unit function is returned.

4.1.3 Differences Between the Lambda and HOL Logics

This section describes the differences between the logics which LAMBDA version 3 uses [57,160], and the logic used by the HOL proof assistant [79]. The latter logic is the same as LAMBDA version 4's logic [58], but there are some differences in the treatment of booleans and truth values. LAMBDA implements a constructive logic whereas HOL uses a classical logic. This entails that we have no law of the excluded middle, and we cannot eliminate double negations.

In HOL all functions must be total. In LAMBDA functions may be partial, which means that for some inputs a function's output may not exist. Non-denoting terms are distinguished from denoting terms through the existence predicate `E`. It is not possible for functions to manipulate truth values in LAMBDA because functions may be partial. Consider the illegal function definition which does not terminate, and would hence be equal to the undefined value of type Ω.

```
fun nondenoting (t: Ω) = (nondenoting t): Ω;
```

How would we deal with the goal $\vdash$ `nondenoting TRUE`? The conclusion has the correct type, but does not equal `TRUE` or `FALSE`. All truth values must be equal to `TRUE` or `FALSE`, so this third value does not make sense. The type *bool* is therefore provided in LAMBDA to manipulate boolean values.

```
datatype bool = true | false;

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

Boolean conjunction `&&`, disjunction `||` are similarly provided in the standard library. As *bool* is an object-level type it may be manipulated by functions:

```
fun nondenoting (t: bool) = (nondenoting t): bool;
```

Now when we use the term `nondenoting true`, it is a boolean value, and we can therefore not write $\vdash$ `nondenoting true`. However, $\vdash$ `E (nondenoting true)` is a legitimate goal, and may be proved to be equal to $\vdash$ `E UNDEFINED` which is `FALSE`. This means that there is no bijection between boolean values `true` and `false` and truth values `TRUE` and `FALSE`. Conceptually we can map `true` to `TRUE` and `false` to `FALSE`, but we cannot map the non-denoting term `UNDEFINED`: *bool* to any truth value. (We say conceptually because object-level functions in LAMBDA cannot manipulate meta-level objects, such as truth values.)

Due to the distinction of boolean values and truth values one cannot write certain expressions in LAMBDA which are legitimate in HOL. Consider an AND gate, which may be defined as follows in HOL.

```
⊢ andgate(x,y,z) = (z = x ∧ y)
```

It is then possible to write

```
⊢ andgate(in,T,out) ∧ T
```

The first `T` is a boolean value on an input of the AND gate. The second `T`, however, is a truth value. `T` is used in two completely unrelated capabilities. Also note that in the definition of the AND gate $\wedge$ is used as the and operator on data values, and in the goal it is used as conjunction on truth values. We would argue that it is unnatural to identify these two different types. In LAMBDA one would write:

```
⊢ andgate#(in,true,out) ∧ TRUE
```

Where `andgate` would be defined as

```
val andgate#(x,y,z) = (z == x && y);
```

Here the two data values and truth values are clearly distinguished by virtue of their types: `true`: *bool* and `TRUE`: Ω. LAMBDA version 4 has the same logic as HOL, but it maintains the distinction between *bool* and Ω. However, in LAMBDA version 4 it is possible to define a bijection between the two types, *e.g.* by mapping `true` to `TRUE` and `false` to `FALSE`.

In both HOL and LAMBDA it is possible to write a function `f` with type `f`: ...`->` *bool*. Only in HOL is it possible to interpret this as a proposition. In LAMBDA this would correspond to changing `f`'s type to `f`: ... `->` Ω. In LAMBDA there is no universal and existential quantification over booleans with result type *bool*. *i.e.* $(\forall x.\ \mathtt{P\#}(x)) : bool$ is an ill-typed expression. The result of a quantification must have type omega. The lack of quantification whilst remaining inside booleans means that we cannot write a useful behaviour function in LAMBDA. Recall from Section 2.2 that a behaviour function maps a circuit expression to a formulae describing its behaviour. In HOL this amounts to writing a function `behaviour`: *circuit* `->` *bool*, because booleans and truth values are equal. In LAMBDA `behaviour`: *circuit* `->` Ω is not allowed, and `behaviour`: *circuit* `->` *bool* is not satisfactory because booleans do not have enough expressive power to describe interesting behaviours.

In HOL Hilbert's operator ε is used to give implicit descriptions [79,58]. $(\varepsilon x :\ t.\ \mathtt{f}\ x) : t$ denotes a member of the set with characteristic function `f`. If this set is empty, an arbitrary member of type t is returned. It is not known which member this is, so that the epsilon expression may be viewed as an irreducible term. It is therefore possible to use (εx. `T`) as a don't know value. However, in LAMBDA `UNDEFINED`, conceptually defined as ιx. `FALSE`, cannot be used as a don't know value because functions are strict.

4.2 Encoding picoELLA in Lambda

In Section 3.3 we described picoELLA's syntax and semantics. In this section we will describe how picoELLA may be embedded in LAMBDA. The basic idea is to use LAMBDA's definitional mechanisms to encode a representation of circuit expressions and a semantics operating on this representation (Section 2.3.) We then prove results we expect to hold of this semantics to gain confidence in the correctness of the embedding. A substantial number of auxiliary definitions are necessary for the dynamic and static semantics. In Sections 4.2.1 and 4.2.2 we define various types which are used, together with some operations on these types. Following this, we give the embedding of the static and dynamic semantics of picoELLA. Finally, we compare our approach to other work.

4.2.1 Constants and Types

The embedded syntax of picoELLA is a subset of the syntax given in Section 3.3. Specifically picoELLA type definitions[1] and the `INPUT` construct are missing. The reason for omitting type definitions becomes clear when we reason about program fragments. An expression with constructors such as `hi` or `lo` is statically typable only in an environment in which these constructors have been declared. As a result we have to attach a type environment to every expression we want to reason about. This becomes very cumbersome, and leads to problems when combining

[1] We will use the term picoELLA type to describe the type when an embedded picoELLA constant or expression would have when typed using the static picoELLA semantics of Section 3.3.2. By the (LAMBDA) type of any LAMBDA expression, including those encoding picoELLA expressions and constants, we indicate the type ascribed to the expression using LAMBDA's type system.

expressions, *e.g.* using `LET` statements or tuples. We have to combine the type environments also, which means that we have to resolve potentially clashing type definitions; `hi` could be declared in two separate enumerated types, for example. Also, if `hi` is the result of an expression, the type of `hi` must also be part of the type environment of the enclosing expression. The solution we have adopted is to attach the type to the constructor, and so make every constructor uniquely typable in every context. However, this gives us less information about a picoELLA type than when using an explicit type environment. We only know the type of the particular constructor we are dealing with, but we don't know anything about possible other constructors of the picoELLA type. This poses no problems until we want to reason about all values of a picoELLA type. We describe this in more detail below. Due to the omission of picoELLA type definitions we cannot use tuple types explicitly. The constant `?type` where `type` is a picoELLA tuple type, is unavailable. However, since `(?type1,?type2)` behaves identically to `?type`, where `type = type1 * type2`, we do not lose any expressiveness. A useful result of the omission of explicit tuple types, is that we do not need to retain the definitions of tuple types at run time. Recall that in the dynamic semantics `?type` evaluates to itself, if `type` is an enumerated type, or otherwise to a tuple containing the bottom values of its constituent subtypes. By omitting tuple type definitions we can dispense with dynamic semantics rule 3.39 and retain a monogenic semantics.

All picoELLA constants (see Section 3.3) are encoded by the LAMBDA type *const*, which is defined as follows.

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

A constant is a constructor `Cons(`i`,`t`)`, or a binary tuple containing two constants. `Cons(`i`,`t`)` encodes the ith constructor of type t, with the convention that the zeroth constructor `Cons(0,`t`)` represents the undefined value `?t` of type t. As LAMBDA implements a logic of partial terms, we might want to use the undefined value `UNDEFINED` to encode bottom constants. However, `UNDEFINED` is strict in the sense that, for example `(CoTuple (UNDEFINED, Cons (1,1))) [2]`, is equal to `UNDEFINED[2]` which is equal to `UNDEFINED`. It should, however, be equal to `Cons (1,1)`. This strictness of undefined also extends to functions, so that `(fn` x `=> true) UNDEFINED` is `UNDEFINED` and not `true`. Whenever an undefined value arises it would propagate through all values and functions, always resulting in an `UNDEFINED` output. Thus `UNDEFINED` is a non-denoting term, whereas `Cons (0,`t`)` does exist and means 'don't know'.

We have one LAMBDA type which encodes picoELLA constants of various picoELLA types, such as `bool` and `bit`.

```
TYPE bit = hi | lo IN TYPE bool = true | false IN ...
```

It is the responsibility of the user to assign a certain interpretation to the encoded constructors. This will normally be done by using the meta-level definition facilities in LAMBDA. If we designate the natural number `1` as representing, for example, the type of bits we can define the LAMBDA constants `bit`, `hi`, and `lo` to stand for the corresponding picoELLA constructors.

```
val bit = Cons(0,1);     (* Meaning ?bit *)
val hi = Cons(1,1);
val lo = Cons(2,1);
```

In Section 4.2.4 we discuss why the don't know value must be encoded as an additional value, and cannot be dealt with otherwise.

We now introduce the LAMBDA type *tpe* encoding picoELLA types, followed by the static semantics for constants, corresponding to rules 3.17, 3.18, and 3.19 of Section 3.3.2.

```
datatype tpe = Type of natural | TyTuple of tpe * tpe;
fun typeOfConst (Cons (_, t)) = Type t |
    typeOfConst (CoTuple (c, d)) =
                TyTuple (typeOfConst c, typeOfConst d);
```

In effect, `typeOfConst` converts `Cons` and `CoTuple` to `Type` and `TyTuple` respectively. The `typeOfConst` function gives us the typing power of the static picoELLA semantics encoded in LAMBDA. Thus, if $S \vdash c : \tau$ in the static picoELLA semantics, then `typeOfConst` d `==` `Type` t holds in the embedding, where d equals `Cons (`n`,`t`)` (for some n and t) and `Type` t are the encodings of c and τ respectively.

A number of operators acting on constants and types have been defined. We have given a number of functions encoding the syntactic equality predicate for various types. For example, `eq` compares two natural numbers.

```
fun eq 0 0 = true |
    eq 0 (S _) = false |
    eq (S _) 0 = false |
    eq (S n) (S m) = eq n m;
```

The function `eq`: *natural* `*` *natural* `->` *bool* encodes equality on natural numbers. The reason for encoding a straightforward equality function explicitly is a historic one. In LAMBDA version 2.0 no boolean equality predicate was provided, so that it had to be defined for every type. In version 2.2 `=` was introduced with type `=`: *"a* `*` *"a* `->` *bool*. Recall that *"a* is an equality type. At this point it would have been too much work to convert all uses of the explicit equality functions to the built-in equality, especially in proofs. We did, however, prove that these equality functions behave identically to the built-in equality. For example:

```
⊢ ∀ n,m.  (eq n m = true) === (n == m)
```

From this we can derive that the equality functions are reflexive, commutative, and transitive. The definition for constant equality is equally straightforward.

```
fun ceq (Cons (c,s)) (Cons (d,t))  = eq s t && eq c d |
    ceq (Cons _) (CoTuple _)       = false |
    ceq (CoTuple _) (Cons _)       = false |
    ceq (CoTuple (c1,c2)) (CoTuple (d1,d2)) =
                                     ceq c1 d1 && ceq c2 d2;
```

Note that `&&` is the infix boolean conjunction, and that we rely on the previously defined `eq` to compare natural numbers. A function implementing type equality `typeEq` is defined in a similar manner.

We will now define a function implementing the data ordering on constants `cle`, and its extension to lists of constants `lle`.

```
fun cle (Cons (c,s)) (Cons (d,t)) =
                     eq s t && (eq 0 c || eq c d) |
    cle (CoTuple (c1,c2)) (CoTuple (d1,d2)) =
                     cle c1 d1 && cle c2 d2;
fun lle nil _ = true |
    lle (_::_) nil = false |
    lle (h1::t1) (h2::t2) = cle h1 h2 && lle t1 t2;
```

The definition of `cle` is partial, and is defined only on constants of the same picoELLA type. Recall that `Cons(0,`*t*`)` is the bottom value of type *t*. `lle` is a total function because an empty list is less than or equal to any other list. For example, `lle [`*c*`] [`*c*`,`*d*`] = true` because `lle [] [`*d*`] = true` and `cle` *c c* `= true`. If we interpret a list of constants as an environment or stack this corresponds to a natural ordering on environments. Finally we define the greatest lower bound function, followed by the projection of a constant to its bottom value ($\downarrow$ of Section 3.1).

```
fun glb (Cons c) (Cons d) =
        if cle (Cons c) (Cons d) then
               Cons c
        else if cle (Cons d) (Cons c) then
               Cons d
        else
               bottomOfConst (Cons c) |
    glb (CoTuple (c1,c2)) (CoTuple (d1,d2)) =
        CoTuple (glb c1 d1, glb c2 d2);
```

```
fun bottomOfConst (Cons (_, t)) = (Cons (0, t)) |
    bottomOfConst (CoTuple (c, d)) =
                  CoTuple (bottomOfConst c, bottomOfConst d);
```

Appendix B contains a list of basic results proved in LAMBDA about various operators, such as `cle` and `glb`'s totality, reflexivity, and transitivity. Various results relating `ceq`, `cle`, `glb`, *etc.* have also been proved.

Constant Induction Principles

We will now briefly discuss reasoning about constants of one picoELLA type. The induction principle which LAMBDA returns for the *const* data type is:

```
***** Premise 2 *****
1:  E r1'
2:  E r'
1:  Pconst#(r1')
2:  Pconst#(r')
⊢ Pconst#(CoTuple (r1',r'))
***** Premise 1 *****
1:  E r3'
2:  E r2'
⊢ Pconst#(Cons (r3',r2'))
--------------------
1:  E w
⊢ Pconst#(w)
```

When dealing with functions such as `typeOfConst` and `cle`, which destruct the `Cons` constructor we need a more discerning induction principle. For constants which are enumerated types, we have to deal with all possible types, and all possible constructors. However, in all functions which are used in the embedding all enumerated types are treated uniformly so that it is not necessary to do an induction on the *r2'* variable in the first premise. However, the zeroth constructor is treated differently from the remaining constructors, so that it is useful to do a natural number induction on *r3'*. In the inductive case the induction hypothesis is generally useless because all non-bottom values are treated in the same manner in function definitions. Functions operating on constants are not defined on constructor number, but only use the distinction between bottom values and non-bottom values.

We would like to have the following induction principle, based on the data ordering `cle`.

```
***** Premise 2 *****
⊢ P#(⊥)
**** Premise 1 *****
∀d.  cle d c == true → P#(d) ⊢ P#(c)
---------------------
⊢ ∀c.  P#(c)
```

It is similar, though not identical, to a fixed point induction [118]. To try to prove this rule we define a measure `sizeOfConst` on constants.

```
fun sizeOfConst (Cons (0, _)) = 1 |
    sizeOfConst (Cons (S _, _)) = 0 |
    sizeOfConst (CoTuple (x, y)) = sizeOfConst x + sizeOfConst y;
```

The size of a constant is the maximum number of steps it is away from being a fully defined value, where a step corresponds to changing a single bottom value to a constructor. This measure gives rise to an induction principle which is less useful than expected. The reason for this is that
`sizeOfConst` c `<` `sizeOfConst` d $\rightarrow$ `cle` c d does not hold. As a counter example, take c to be `CoTuple (Cons (0,`t`), CoTuple (Cons (0,`t`), Cons (1,`t`)))` and d to be `CoTuple (Cons (1,`t`), CoTuple (Cons (1,`t`), Cons (0,`t`)))`. It is not possible to encode a `sizeOfConst` function which has this property, because it would have to linearise (map into natural numbers) an inherently non-linear

ordering (a semi-lattice.) We therefore intend to use `sizeOfConst` only in conjunction with the `cle` operator. Note that `sizeOfConst` as defined above, is not strictly increasing, *i.e.*

$\exists c, d.$ `sizeOfConst (CoTuple (`c`,`d`))` $\not>$ `sizeOfConst` c `+ sizeOfConst` d.

This means that it cannot be used by itself as the basis of a useful induction principle. We can fix this trivially, by adding one to the size of any tuple, but since we use `sizeOfConst` in conjunction with `cle` only, we will keep this conceptually cleaner version.

We can only reason about constants of all picoELLA types, unless we qualify the statement by using `typeOfConst`. For example,

$\vdash \forall c.$ `typeOfConst` c `==` $t \rightarrow$ `P#(`c`)`. Even in this case we do not know anything about the number of constructors in the picoELLA type. We can use an explicit assumption to define the constructors.

```
val BITINDUCT = ∀c.  typeOfConst c == bittype →
                                c == hi ∨ c == lo ∨ c == bit;
```

(`bittype` is defined to be `Type 1`, say.) Sometimes we want to restrict our attention to defined values of a type only.

```
val DEFBITINDUCT = ∀c.  typeOfConst c == bittype →
                                   c == hi ∨ c == lo;
```

With one of these abbreviations as an hypothesis we can instantiate c with the variable we want to do induction over, and define the derived `bitInduct` rule.

```
⊢ Pbit#(bit)
⊢ Pbit#(lo)
⊢ Pbit#(hi)
--------------------
E w, typeOfConst w == bittype ⊢ Pbit#(w)
```

Often we want to reason about two constants which have the same picoELLA type, although we are not interested in the exact type. We can use the following derived induction rule `twoConstInduct` for this.

```
***** Premise 2 *****
1:  E r7'
2:  E r5'
3:  E r4'
⊢ P#(Cons (r5',r4'),Cons (r7',r4'))
***** Premise 1 *****
1:  E r3'
2:  E r2'
3:  E r1'
4:  E r'
1:  P#(r1',r3')
2:  P#(r',r2')
3:  typeOfConst r1' == typeOfConst r3'
4:  typeOfConst r' == typeOfConst r2'
⊢ P#(CoTuple (r1',r'),CoTuple (r3',r2'))
--------------------
1:  E c
2:  E d
1:  typeOfConst c == typeOfConst d
⊢ P#(c,d)
```

This is a good example of the use of derived rules. We could use a tactic of the form (the actual tactic is more involved):

```
doRules [allR,constInduct,allR,constInduct] thenT typeOfConstTac
```

but it may have unwanted side effects, apart from duplicating work. The application of tactic `typeOfConstTac` will rewrite the whole premise it is applied to, whereas the derived rule will only change the variables c and d in the context `P`. The application of this tactic applies four rules as well as the rewrite rules of `typeOfConstTac`, whereas `twoConstInduct` is a single rule.

4.2.2 Expressions

We will now define the type of picoELLA expressions, and some auxiliary functions operating on expressions. The expression type contains two auxiliary types, *const*, which we have already seen, and *chooser*.

```
datatype chooser = C of const |
                   B of chooser * chooser |
                   T of chooser * chooser;
```

The chooser data type encodes the syntactic category of choosers. It is different from the paper syntactic definition because it uses the type of constants instead of duplicating them (Section 3.3.) This allows the chooser `C (Cons (0,`*type*`))`. Following the convention that the zeroth constructor encodes the undefined value of the type, this would be a term which cannot be produced in the paper syntax. We therefore interpret this chooser as the wild card chooser of its type. Thus `Cons (0,`*type*`)` stands for the constant `?type`, and `C (Cons (0,`*type*`))` for the chooser `type`. This gives an intuitive dual interpretation. Typing of choosers, corresponding to static semantics rules 3.19, 3.20, 3.21, and 3.22 is defined in the embedding as follows.

```
fun typeOfChooser (C c) = (typeOfConst c, true) |
    typeOfChooser (B (ch1, ch2)) =
          (fn (t1, b1) =>
          (fn (t2, b2) =>
                    (t1, b1 && b2 && typeEq t1 t2)))
                    (typeOfChooser ch1) (typeOfChooser ch2) |
    typeOfChooser (T (ch1, ch2)) =
          (fn (t1, b1) =>
          (fn (t2, b2) =>
                    (TyTuple (t1, t2), b1 && b2)))
                    (typeOfChooser ch1) (typeOfChooser ch2);
```

This definition would normally be written in ML using
`let val` $(t1, b1)$ `= typeOfChooser` $ch1$ `in` ..., but this is not allowed in the ML subset which is used in LAMBDA. The typing of choosers is more involved than the typing of constants because they are not necessarily well-typed. There is a choice in how we handle this. We could make the function partial on those choosers which are not well-typed, which would correspond naturally to the fact that in the paper static semantics there is no derivation for the type. However, it is easier to return a pair of values than it is to reason about undefined terms. The second component of the pair indicates success or failure, the first the type of the chooser if the derivation is successful. If the derivation is not successful the type is irrelevant. Even using this solution we have a choice of what we return when the chooser is ill-typed. Do we always return the same type, or do we salvage as much of an incorrect type as possible? In the solution above, we always return the first type of a bar chooser, even if the types of the first and second component don't agree. This means that the following result does not hold.

$$\vdash \forall x, y.\ \texttt{typeOfChooser (B (}x, y\texttt{)) == typeOfChooser (B (}y, x\texttt{))}$$

If x and y are both well-typed, but have differing types, then the type component of the result type will be different on the left and right hand side of the equality. What we can prove is the following:

$$\begin{aligned}&\vdash\ \forall x, y, t.\\ &\quad \texttt{typeOfChooser } x \texttt{ == (}t\texttt{,true)} \wedge \texttt{typeOfChooser } y \texttt{ == (}t\texttt{,true)} \rightarrow\\ &\qquad\quad \texttt{typeOfChooser (B (}x, y\texttt{)) == typeOfChooser (B (}y, x\texttt{))}\end{aligned}$$

If would have been possible to obtain the first statement, but it would have complicated the definition of `typeOfChooser` without any other advantage. Like constant equality `ceq` chooser equality `cheq` is defined so that it too is equal to the boolean equality `=`. It is easy to define a chooser normal form, and functions mapping a chooser onto its normal form. It turns out to be hard to reason about these functions, however, and no significant results have been proved about the chooser normal form.

We can now define the data type representing expressions of LAMBDA type *expr*, or circuit.

```
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 * chooser |
                Index1 of expr |
                Index2 of expr |
                LetRec of const * expr * expr;
```

We justified the absence of the type declaration constructor `TYPE` earlier in this section. We decided that a uniform treatment of variables was important because we would be reasoning more often about program fragments than complete programs. For this reason the `INPUT` construct has been omitted and any input variables are regarded as free variables in the expression. They receive their values through the enclosing value environment, as we shall see in the embedded dynamic semantics.

We use the de Bruijn encoding [48] for lambda abstractions. This removes the need to formalise variable names. In the de Bruijn encoding variable names are replaced by a natural number indicating the distance to the defining lambda, measured in intervening lambdas. The value environment then becomes a *natural* to *const* mapping, which we may implement as a stack. For example, the picoELLA fragment

```
LET x = a IN (x, LET y = b IN (x, y))
```

would be given in the embedded syntax as

```
Let (a, Tuple (Var 0, Let (b, Tuple (Var 1, Var 0))))
```

We would have preferred to use LAMBDA's meta-variables to encode abstraction instead of implementing it at the object level. Recall from Section 4.1 that abbreviations are syntactic meta-functions, which map an object-level expression to another object-level expression. A `LET` expression would have been written either as

```
Let (e, lam x.  f)                                                    (1)
```

or, if `F#(`x`) =` f, as

```
Let (e, F#(x))                                                        (2)
```

In the former expression the `lam` constructor is an unnamed syntactic lambda abstraction (*cf.* an object-level lambda abstraction `fn` x. f.) The second expression uses a named syntactic lambda abstraction, *i.e.* an abbreviation. The great advantage of this method over implementing abstraction at the object level is that issues such as variable capture and scoping are dealt with by the proof system. A severe drawback is that it would not have been possible to reason about the abstraction mechanism, because it is not part of the logic. However, LAMBDA does not allow users access to the syntactic lambda abstraction constructor `lam` [64, Sections 2.4.1 and 2.6.1]. In logic frameworks such as the Edinburgh LF [89] it is possible to take this approach because the logic is defined explicitly by the user and meta-level constructors such as `lam` must be available. Another disadvantage of using meta-level lambda abstraction is that it implements call-by-name, rather than call-by-value [119]. In terms of hardware this means that in `(1)` and `(2)` above the defining expression would have been replicated, instead of implementing a fan-out. Failing this approach we could have approximated the use of the syntactic lambda abstraction (function type) by the object-level function arrow.

```
datatype expr = Let (expr, expr -> expr) | ...
```

`->` is the function type at the object level, rather than the meta-level. Due to possible inconsistencies in the LAMBDA version 2 logic the use of function spaces was restricted, so that the type being defined could not appear on the left hand side of the function arrow [61]. We can get around this by not passing the expression in to the abstraction, but the evaluated expression. This is more in line with what we would like to do anyway, because the defining expression is then evaluated once, no matter how many times a reference to it occurs in the using expression. As mentioned previously, this corresponds to a fan-out, rather than replication of the defining hardware. This would be accomplished by the definition

```
datatype expr = Let (expr, const -> expr) | ...
```

In this case, the function expression would have to be a lambda term of the form `fn` x `=>` e, or a previously defined function. However, in both cases the totality of the functional expression is a prerequisite for the totality of the static and the dynamic semantics. This would have been quite cumbersome to reason about. (Meta-level lambda expressions would always have been total.) A more serious drawback is that the induction principle returned by LAMBDA is not strong enough. The premise dealing with the `LET` would be as follows:

```
E r1', E r', Pexpr#(r1') ⊢ Pexpr#(Let (r1',r'))
```

Thus we get no induction hypothesis for the result expression of the function *r'*. For these reasons we chose to use the de Bruijn encoding. In the related work Section 4.2.4, we will compare this aspect of our approach to other research.

Recursive `LET` statements are encoded by the `LetRec` constructor, which carries its initial approximation with it (see dynamic semantics rules 3.28 and 3.29 of Section 3.3.3.) This makes the typing of expressions monogenic because the type of the initial approximation is unique, and the defining expression must have the same type (rule 3.10 and comments on page 70.)

We define a function mapping expressions to their size, which is a natural number. It is a strictly increasing function, in that the size of an expression is strictly greater than the size of each of its subexpressions. This allows us to use this measure as the basis for an induction principle on expressions.

```
fun sizeOfExpr (Const c) = sizeOfConst c + 1 |
    sizeOfExpr (Tuple (e, f)) = sizeOfExpr e + sizeOfExpr f |
    sizeOfExpr (Let (e, f)) = sizeOfExpr e + sizeOfExpr f |
    sizeOfExpr (Var n) = S n |
    sizeOfExpr (Delay (c, e)) = sizeOfConst c + 1 + sizeOfExpr e |
    sizeOfExpr (If (e1, e2, e3, ch)) =
               sizeOfExpr e1 + sizeOfExpr e2 + sizeOfExpr e3 |
    sizeOfExpr (Index1 e) = sizeOfExpr e + 1 |
    sizeOfExpr (Index2 e) = sizeOfExpr e + 1 |
    sizeOfExpr (LetRec (c, e, f)) =
               sizeOfConst c + 1 + sizeOfExpr e + sizeOfExpr f;
```

Note that the size of a `Var` is how far it reaches into the environment (plus one to ensure that `sizeOfExpr` is strictly increasing.) It is possible to return any fixed constant value, but this complicates `reduce`'s mononicity theorem (Section 4.3.2) because the induction hypothesis cannot be used for `Var` constructs in this case.

4.2.3 The Embedded Static and Dynamic Semantics

The static semantics of program expressions is simplified by not embedding tuple types, as we explained in Section 4.2.1. Moreover, by insisting on an initial approximation for recursive `LET` statements we can type expressions unambiguously. The function `typeOfExpr` implements the static semantics, and like `typeOfChooser`, returns a pair of values. The second element indicates whether the expression is well-typed or not. The first element returns the type of well-formed expressions. All of the clauses follow the same pattern, so we will discuss only the most interesting clause. The type of the function is
`typeOfExpr:` *tpe list* `->` *expr* `->` `(`*tpe* `*` *bool*`)`. The type environment acts as a stack onto which newly defined types are pushed. This means that the type of the most recently declared variable is accessed as `Var 0` in the encoding (there

are no intervening lambdas.) All previously declared variables are now accessed as `Var (S n)`, where previously they were accessed as `Var n`.

```
fun typeOfExpr te (Const c) = (typeOfConst c, true) |
    typeOfExpr te (Tuple (e1, e2)) =
          (fn (t1, b1) =>
          (fn (t2, b2) =>
                   (TyTuple (t1, t2), b1 && b2)))
                   (typeOfExpr te e1) (typeOfExpr te e2) |
    typeOfExpr te (Let (e1, e2)) =
          (fn (t1, b1) =>
          (fn (t2, b2) =>
                   (t2, b1 && b2))
                   (typeOfExpr (t1::te) e2)) (typeOfExpr te e1) |
    typeOfExpr nil (Var 0) = (Type 0, false) |
    typeOfExpr nil (Var (S n)) = (Type 0, false) |
    typeOfExpr (_::t) (Var (S m)) = typeOfExpr t (Var m) |
    typeOfExpr (h::_) (Var 0) = (h, true) |
    typeOfExpr te (Delay (c, e)) =
          (fn (t, b) =>
                  (t, b && (typeEq t (typeOfConst c))))
                  (typeOfExpr te e) |
    typeOfExpr te (Index1 e) =
          (fn (TyTuple (t, _), b) => (t, b) |
              (Type _, _) => (Type 0, false)) (typeOfExpr te e) |
    typeOfExpr te (Index2 e) =
          (fn (TyTuple (_, t), b) => (t, b) |
              (Type _, _) => (Type 0, false)) (typeOfExpr te e) |
```
(continued on next page)

```
(continued from previous page)
    typeOfExpr te (If (e1, e2, e3, ch)) =
          (fn (tc, bc) =>
          (fn (t1, b1) =>
          (fn (t2, b2) =>
          (fn (t3, b3) =>
                   (t2, bc && b1 && b2 && b3 &&
                        (typeEq tc t1) && (typeEq t2 t3))))))
                   (typeOfChooser ch) (typeOfExpr te e1)
                   (typeOfExpr te e2) (typeOfExpr te e3) |
    typeOfExpr te (LetRec (c, e1, e2)) =
          (fn tc =>
          (fn (t1, b1) =>
          (fn (t2, b2) =>
                   (t2, b1 && b2 && (typeEq t1 tc) &&
                        (ceq (bottomOfConst c) c)))
                   (typeOfExpr (tc::te) e2))
                   (typeOfExpr (tc::te) e1))
                   (typeOfConst c);
```

The `LetRec` clause encodes the rules 3.16 and 3.10 of the static semantics. It computes the type *tc* of the constant, which is then also used to type the defining expression. The type *tc* is added to the type environment, which has LAMBDA type *tpe list*. The defining and result expressions return types *t1* and *t2* respectively. The whole `LetRec` is well-typed only if both expressions are well-typed (*b1* `&&` *b2*), the return type of the defining expression is the same as the type of the constant (`typeEq` *tc t1*), and the constant *c* is a bottom value. This last condition is checked by the side condition $\downarrow c = c$ in rule 3.10, which is encoded as `ceq (bottomOfConst` *c*`)` *c*.

The type of an expression is a very crude measure; wildly different expressions can have the same type. Often we want to reason about circuits which do not only have the same type, but have the same 'shape.' For example, we would like to prove that the dynamic semantics preserves the shape of a circuit. The function `shapeEq` encodes shape equality on expressions. This is the only equality function which is less discerning than the boolean equality `=`.

```
fun shapeEq (Const c) (Const d) =
                      typeEq (typeOfConst c) (typeOfConst d) |
    shapeEq (Tuple (e1, e2)) (Tuple (f1, f2)) =
                      (shapeEq e1 f1) && (shapeEq e2 f2) |
    shapeEq (Let (e1, e2)) (Let (f1, f2)) =
                      (shapeEq e1 f1) && (shapeEq e2 f2) |
    shapeEq (Var n1) (Var n2) = eq n1 n2 |
    shapeEq (Delay (c, e)) (Delay (d, f)) =
                      (typeEq (typeOfConst c) (typeOfConst d)) &&
                      (shapeEq e f) |
    shapeEq (If (e1, e2, e3, ch1)) (If (f1, f2, f3, ch2)) =
                      (shapeEq e1 f1) && (shapeEq e2 f2) &&
                      (shapeEq e3 f3) && (cheq ch1 ch2) |
    shapeEq (Index1 e) (Index1 f) = shapeEq e f |
    shapeEq (Index2 e) (Index2 f) = shapeEq e f |
    shapeEq (LetRec (c, e1, e2)) (LetRec (d, f1, f2)) =
                      (typeEq (typeOfConst c) (typeOfConst d)) &&
                      (shapeEq e1 f1) && (shapeEq e2 f2) |
    shapeEq (Const _) (Tuple _) = false |
    shapeEq (Const _) (Let _) = false | ...;
    (* All other combinations of constructors result in false *)
```

Thus `shapeEq` checks whether the two expressions have the same abstract syntax tree. Whenever a constant is encountered (`Const`, `Delay`, and `LetRec`) only the type is checked. This is essential for delays because the value in a delay will probably change over time, but it will keep its type. The constant approximation in recursive `LET` statements need not be the same for both expressions. In the case of `Const` constructors, however, the value will not change. We introduce an ordering `ple` on expressions which may be used on shape equal expressions. By allowing constants to be different (but of the same type) we obtain a more detailed ordering.

```
fun ple (Const c) (Const d) = cle c d |
    ple (Tuple (e1, e2)) (Tuple (f1, f2)) =
                  (ple e1 f1) && (ple e2 f2)
    ple (Let (e1, e2)) (Let (f1, f2)) = (ple e1 f1) && (ple e2 f2) |
    ple (Var _) (Var _) = true |
    ple (Delay (c, e)) (Delay (d, f)) = (cle c d) && (ple e f) |
    ple (If (e1, e2, e3, _)) (If (f1, f2, f3, _)) =
                  (ple e1 f1) && (ple e2 f2) && (ple e3 f3) |
    ple (Index1 e) (Index1 f) = ple e f |
    ple (Index2 e) (Index2 f) = ple e f |
    ple (LetRec (c, e1, e2)) (LetRec (d, f1, f2)) =
                  (cle c d) && (ple e1 f1) && (ple e2 f2);
```

The `match` function implements the matching procedure as defined on page 68. It is used as part of the dynamic semantics. Recall that two constructors match if they are identical, and do not match otherwise. A bottom value only matches with the wild card chooser, and delivers neither a match nor a no-match otherwise. This reflects the intuition that a don't know value cannot choose between two constructors. A three-valued boolean logic is used in the matching which may result in a match `tt`, a non-match `ff`, or neither `uu`.

```
datatype bool3 = tt | ff | uu;
```

The functions `and3` is defined as follows.

```
fun and3 uu uu = uu | and3 uu tt = uu | and3 uu ff = ff |
    and3 tt uu = uu | and3 tt tt = tt | and3 tt ff = ff |
    and3 ff _ = ff;
```

Functions `or3` and `not3` are defined similarly. The flat data ordering in which `uu` is the least element, and `tt` and `ff` are less than or equal to themselves but incomparable to each other is implemented by `le3`.

```
fun match (C (Cons (0, s))) (Cons (_, t)) = (eq3 s t) |
    match (C (Cons (S n, s))) (Cons (S m, t)) =
                   and3 (eq3 s t) (eq3 n m) |
    match (C (Cons (S n, s))) (Cons (0, t)) = and3 (eq3 s t) uu |
    match (C (CoTuple (c1, c2))) (CoTuple (d1, d2)) =
                   and3 (match (C c1) d1) (match (C c2) d2) |
    match (B (c1, c2)) d = or3 (match c1 d) (match c2 d) |
    match (T (c1, c2)) (CoTuple (d1, d2)) =
                   and3 (match c1 d1) (match c2 d2);
```

This function is defined only on equally typed choosers and constants. `match` could have been more partial than it currently is, by returning `UNDEFINED` instead of `ff` for unequally typed constructors. The function `eq3` is the same as the embedded equality on natural numbers `eq`, but returns `tt` and `ff` instead of `true` and `false` respectively. The convention that `Cons (0,`*type*`)` represents `?type`, and `C (Cons (0,`*type*`))` represents `type` is implemented here. These special cases account for the messy first three clauses of the function definition.

The function `elem`, used by `reduce`, implements the lookup of variables in an environment. It is a partial function with the following definition.

```
fun elem (h::_) 0 = h |
    elem (_::t) (S n) = elem t n;
```

Like the matching function, the reduction function which implements the dynamic semantics operates only on well-formed expressions.

```
fun reduce l (Const c) = (c, Const c) |
    reduce l (Tuple (e1, e2)) =
             (fn (c1, f1) =>
             (fn (c2, f2) =>
                      (CoTuple (c1, c2), Tuple (f1, f2))))
                      (reduce l e1) (reduce l e2) |
    reduce l (Let (e1, e2)) =
             (fn (c1, f1) =>
             (fn (c2, f2) =>
                      (c2, Let (f1,f2)))
                      (reduce (c1::l) e2)) (reduce l e1) |
    reduce l (Var n) = (elem l n, Var n) |
    reduce l (Delay (c, e)) = (c, Delay (reduce l e)) |
    reduce l (If (e1, e2, e3, ch)) =
             (fn (c1, f1) =>
             (fn (c2, f2) =>
             (fn (c3, f3) =>
                      (case match ch c1 of
                            uu => bottomOfConst c2 |
                            tt => c2 |
                            ff => c3,
                       If (f1, f2, f3, ch)))))
                      (reduce l e1) (reduce l e2) (reduce l e3) |
    reduce l (Index1 e) =
             (fn (CoTuple (c, _), f) => (c, Index1 f))
             (reduce l e) |
```
(continued on next page)

```
(continued from previous page)
    reduce l (Index2 e) =
              (fn (CoTuple (_, c), f) => (c, Index2 f))
              (reduce l e) |
    reduce l (LetRec (c, e1, e2)) =
              (fn (d1, f1) =>
              (fn (d2, f2) =>
                       (d2, LetRec (c, f1, f2)))
                       (reduce (d1::l) e2)) (iterate l e1 c)

and iterate l e c =
              (fn (d, f) =>
                      case ceq c d of
                           true => (d, f) |
                           false => again l e d)
                      (reduce (c::l) e)

and again l e d = iterate l e d;
```

This definition contains a number of points of interest. First note that it depends on `typeOfExpr` to remove ill-formed circuits such as `Index1 (Const (Cons ...))`. The environment is used in the same manner as `typeOfExpr` does, but this time is a value environment, containing constants. The relation between the value environment *l* and type environments *te* is *te* `== map typeOfConst` *l*.

The function `iterate` is used to implement the fixed point computation. Its arguments are the current value environment, the defining circuit, and the previous approximation. By virtue of the static semantics the first invocation of `iterate` will be with a bottom value. The function `again` is purely a technical device to allow rewrite tactics to be written which make the rewriting converging. This will be clearer when we describe some of the tactics which have been written for these functions. See Section 5.1.4 for more details. An alternative definition for `reduce`, implementing the fixed point iteration directly could be given as follows.

```
fun reduce l (LetRec (c, e1, e2)) =
             (fn (d1, f1) =>
                 if ceq c d1 then
                    (fn (d2, f2) =>
                        (d2, LetRec (bottomOfConst c, f1, f2)))
                        (reduce (d1::l) e2)
                 else
                    reduce l (LetRec (d1, e1, e2)))
             (reduce (c::l) e1) | ...
```

We did not take this approach because it is cleaner to reason about the fixed point computation as a separate function. Note that if we did not apply `bottomOfConst` to c, the initial approximation in the result expression would be the fixed point, not the bottom value. Alternatively we could take the result expression apart every time we came out of a recursion, and replace the approximation by the value passed into the recursion. This would correspond to the alternative semantic rules 3.40 and 3.41 of Chapter 3. The `reduce` and `iterate` functions are mutually recursive, and a property of one will involve a similar property of the other.

The function `reduce` implements the part of the dynamic semantics dealing with evaluation within one time step. Time is added by `reduceSeq`, corresponding to semantic rules 3.23, 3.24, and 3.26.

```
fun reduceSeq l e [] = ([], e) |
    reduceSeq l e (h::t) =
              (fn (c1, f1) =>
              (fn (c2, f2) => (c1::c2, f2))
              (reduceSeq l f1 t)) (reduce (h::l) e);
```

We have an initial environment l, onto which the input value for every time step is pushed. The expression result from a `reduce` evaluation is used as the input for the next `reduceSeq` evaluation.

We would preferred to have used a relational approach rather than a functional one. In earlier versions of LAMBDA we could have written

```
val reduce =
    ιR. (∀c.  R (Cons c) (c, Const c)) ∧
        (∀e1,e2,c1,f1,c2,f2.
                R e1 (c1,f1) ∧ R e2 (c2,f2) →
                R (Tuple (e1,e2))
                  (CoTuple (c1,c2), Tuple (f1,f2))) ∧ ...;
```

In other words, the reduction relation is the (smallest) relation satisfying the reduction clauses for the individual constructs. As mentioned in Section 4.1.1, restrictions on the iota operator disallowed this style of definition, and the iota operator is not user-accessible any more.

4.2.4 Related Work

We justified our use of the de Bruijn encoding on page 115. Recall from Section 2.5 that Melham formalises variables as sequences of characters, and defines a valuation function *e: str* `->` *bool* representing the environment [124]. He uses strings *str* to index the environment instead of natural numbers. The semantic function maps circuit expressions to assertions about the environment. For example, the clause `Sm (pwr` *p*`)` *e* `= (`*e p* `= T)` expresses that the entry for *p* in the environment must be `T`. The semantics defines the environment in terms of constraints which are deduced from circuit structure. In our case, the environment is used solely to store intermediate results from `LET` and `LET REC` constructs. It is used purely as a stack.

Some aspects of the embedding in HOL of an ELLA subset by Boulton *et al.* [19,17] have been reviewed in Section 2.2.1. Although it is a partially formal behaviour function, a comparison with our work is useful. The HOL ELLA subset is much larger than picoELLA. ELLA type and function definitions are included, and are translated to HOL type and function declarations. Consider the picoELLA type *bit*:

```
TYPE bit = hi | lo IN ...
```

In our embedding we do not explicitly deal with picoELLA type definitions, but we represent `hi` by `Cons(`i`,`t`)` and `lo` by `Cons(`j`,`t`)`, for some t and $i \neq j \neq 0$. The don't know value `?bit` is encoded by `Cons(0,`t`)` (see Section 4.2.1.) The don't know value is an extra value, declared implicitly, as it is in the original ELLA. In HOL ELLA type definitions use the process of lifting to deal with the don't know values of a type. There is one value `UU` which represents the don't know value for all ELLA types. The type `bit` would be represented by the type *(bit) lifted* in HOL. If the elements of *bit* are `hi` and `lo`, the elements of *(bit) lifted* are `LIFT hi`, `LIFT lo`, and `UU`. The function `UNLIFT` projects lifted values to their unlifted equivalent. The term `UNLIFT UU` is irreducible. `LIFT_apply1` lifts a function:

```
⊢ LIFT_apply1 f x = (x = UU ⇒ UU | LIFT (f (UNLIFT x)))
```

Thus a major difference between the HOL embedding and ours is that in HOL every picoELLA type is represented by a different HOL type, whereas we represent all picoELLA types by the same LAMBDA type *const*.

In our embedding constants may be converted to choosers by applying the `C` constructor. The constant `?type` (`Cons(0,`*type*`)`) is interpreted as the `type` chooser (`C (Cons(0,`*type*`))`.) We have a uniform treatment for constructors such as `hi`; they are the same as constants, choosers (we apply the constructor `C`), and as expressions (`Const` is the conversion operator.) Boulton takes another approach: constants and choosers are represented in the same manner, but differently from constants used as expressions (called time independent units in [17, Section 2.8.1].) We mentioned in Section 3.1 that in version 3.0 ELLA the constant, chooser, and expression syntactic classes all contain constructor names. From ELLA version 4.0 onwards the syntactic classes for constants and choosers were combined. As Boulton's embedding is based on version 4.0, and ours on version 3.0 ELLA this may account for the different outlook on this issue. (To avoid confusion we will use constant, chooser, and time independent unit in the remainder of this paragraph.) Time independent units are represented by time-to-value functions, with an unchanging output [17, Section 2.8.1]. A time independent unit c is mapped onto `SIGNAL` c, where `SIGNAL` c t `=` c. This is a function which returns c for all times t. We explicitly encode the process of matching in the function `match`, whilst in HOL choosers are the active agents in matching. Choosers are represented by functions which, when given a constant, return a lifted boolean [17, Section 2.6], representing a match, no-match, or 'can't decide.' Our three-valued booleans `tt`, `ff`, and `uu` correspond to `LIFT T`, `LIFT F`, and `UU` respectively. We would write `match` ch c, where HOL ELLA has ch c. Thus, when a constructor c is interpreted as a chooser, its semantics is `CONST` c, where `CONST` is defined as

$$\vdash \forall c.\ \ \texttt{LIFT\_apply1}\ (\lambda\ x.\ \ x\ \texttt{=}\ c)$$

A constructor chooser is translated to a function which returns `UU` if its argument is `UU`, `LIFTED T` if its argument is equal to the lifted constructor, and `LIFTED F` otherwise. A type name chooser such as `bit` is mapped onto `MATCH_ALL`, which is defined as

$$\vdash \forall x.\ \ \texttt{MATCH\_ALL}\ x\ \texttt{=}\ \texttt{LIFT T}$$

Finally, the bar chooser (tuple chooser) is implemented as repeated application of the `chooser_OR` (`chooser_AND`) function, which is comparable to our `or3` (`and3`). Constants are used to provide the initial values for delays *etc.* The semantics for the delay statement is therefore complicated by the fact that constants and choosers are treated uniformly in HOL ELLA.

We might want to represent the don't know value not as an additional value, but as an undefined value. By an undefined value we do not mean a non-denoting value such as `UNDEFINED`, but a value of which we only know that it is equal to an (unknown) constructor. (This approach was described as the *set semantics* in Section 3.3.5, and Harrison's ELLA embedding in HOL [18, Section 7.8] on page 81.)

In LAMBDA the variable *undefbit* together with the hypothesis *undefbit* `==` `hi` $\vee$ *undefbit* `==` `lo` would do. Note that *undefbit* `===` $(\iota x.\ \ x$ `==` `hi` $\vee$ x `==` `lo`) would make *undefbit* equal to `UNDEFINED`, which is not acceptable. In HOL the Hilbert operator provides the wanted intuition: $(\varepsilon x.x =$ `hi` $\vee\ \ x =$ `lo`$)$ is either `hi` or `lo`, but we cannot prove that it is either. Thus we can prove the following lemma in HOL:

undefbit `=` $(\varepsilon x.\ \ x$ `=` `hi` $\vee$ x `=` `lo`$)$ $\vdash$ *undefbit* `=` `hi` $\vee$ *undefbit* `=` `lo`

When we match *undefbit* with the chooser `hi|lo` we obtain `LIFT T`. This is incorrect; it should be `UU`. In picoELLA LAMBDA terms, this would deliver `tt`, but should deliver `uu`. With `?bit` encoded as *undefbit* the choosers `bit` and `hi|lo` have been identified, which is incorrect. This proves that `?type` really is a distinct value, declared implicitly for every ELLA type.

Another consequence of using *undefbit*, is that all don't know values are the same. The ELLA fragment

```
CASE (?bit,?bit) OF (hi,hi)|(lo,lo):  hi ELSE lo ESAC
```

and its corresponding picoELLA

```
IF (?bit,?bit) MATCHES (hi,hi)|(lo,lo) THEN hi ELSE lo
```

would output `hi` rather than `lo`. The reason for this is that we can prove that `(?bit,?bit)`, which is represented by `(`*undefbit*`,`*undefbit*`)`, is equal to either `(hi,hi)` or `(lo,lo)`. A case analysis selects the `THEN` branch in both cases, so that the output is `hi`. The problem here is that all don't know values are considered to be equal because of the reflexivity of equality. In ELLA and picoELLA, however, every don't know value is different, even in the following case:

`LET` x `= ?bit IN (`x`,`x`)`

It is therefore not possible to represent the tuple by `(`*undefbit*`,`*undefbit*`)` in LAMBDA. We have to use two different don't know values: `(`*undefbit1*`,`*undefbit2*`)`, where both are defined as *undefbit* previously. This also cures the incorrect output above. Of course, this becomes very cumbersome as we need to generate a new variable *undefi* for every `IF` and `LET REC` statement, as both can result in a bottom value. This solution does not work in HOL, because

$\vdash$ *undefi* = $\varepsilon x.\quad x$ = hi $\vee$ x = lo

still allows us to prove that *undefi* = *undefj* for all i and j.

4.3 Results About the Embedding

By proving results which we expect to hold about the embedded semantics we can increase our confidence in the embedding. The dynamic semantics, for example, defines a total function, but this is not obvious from the definition of the `reduce` function. A number of lemmas have to be proved about auxiliary functions before we can attempt this proof. Reflexivity, commutativity, transitivity, totality, and monotonicity are examples of properties we are interested in. More than 700 results have been proved, ranging from simple rewrite rules and lemmas to the complex monotonicity theorem of the dynamic semantics. The proof scripts, containing more than 14,000 lines with over 10,000 rule applications[2] and 2,800 tactic applications, take nearly 13 hours to run on an unloaded sun 4/65 (sparc station 1+) with 32M memory and a local hard disk. We will discuss only a fraction of this effort. Appendix B describes the most interesting results which have been proved, and presents some statistics about the proofs. In this section we will explain the proof of the totality and monotonicity of the `match` and `reduce` functions. Some corollaries following from these results will also be given.

[2]These are derived rule applications, not basic inference rules. Rewrite rule applications are also not included. LAMBDA does not provide any tools to count the number of basic inference rules which have been applied.

4.3.1 Totality and Monotonicity of Matching

The process of matching a chooser and a constant was defined on page 68 in the 'paper semantics', and on page 123 in the embedded semantics. The totality and monotonicity of the matching process depend on the corresponding properties of the `and3` and `or3` functions which are used by `match`. The totality of `match` was proved first, and is straightforward though tedious. The function `match` operates on a well-typed chooser and a constant of the same type.

$$\boxed{\vdash \forall ch, c.\ \ \texttt{typeOfChooser}\ ch\ \texttt{==}\ (\texttt{typeOfConst}\ c,\texttt{true}) \rightarrow \texttt{E}\ (\texttt{match}\ ch\ c)}$$

The proof starts with a structural chooser induction on *ch*. This results in a subgoal for each of the different types of choosers: constant chooser, bar chooser, and tuple chooser.

Case (`C` *d*) We now do a constant induction on *d*, followed by a constant induction on *c*. By using the assumption that the chooser and constant *c* have the same type we discharge two of the four subgoals by rewriting. This leaves us with the `match (C (Cons (`*i*`,`*t*`)))` `(Cons(`*j*`,`*t*`))` and `match (C (CoTuple (`*c1*`,`*c2*`)))` `(CoTuple (`*d1*`,`*d2*`))` cases. The latter subgoal is proved by using the induction hypotheses. The former is more complicated due to our encoding of the bottom element `Cons (0,`*t*`)` and the wild card chooser `C (Cons (0,`*t*`))`. We have to do a natural number induction on both *i* and *j*, producing the three distinct clauses of the matching algorithm. (See the definition of `match` on page 123.) All three cases are dealt with by rewriting using the definition of `match`, followed by the use of the totality of `eq3` and `and3`.

Case (`B (`*ch1*`,`*ch2*`)`) This subgoal is proved by applying the induction hypotheses, and using the totality of `or3`.

Case (`T (`*ch1*`,`*ch2*`)`) A constant induction on *c*, followed by using the fact that *c* must be a tuple type leaves one subgoal: matching a tuple chooser with a tuple constant. The induction hypotheses together with the totality of `and3` then proves this subgoal.

This proof takes nine tactic applications. The following tactic, proving the `(C (CoTuple (`$c1$`,`$c2$`)))` subgoal above, is typical.

```
applyTac ((doRule andL) thenH [2] thenRL [andL,eqCommL]
         thenH [6] thenRL [allL,typeOfConstTotalL,eqReflL,imp2L']
         thenH [8] thenG [2] thenRL [allL,typeOfConstTotalL,
         eqReflL,imp2L'] thenH [3,1] thenT (htogTac 2) thenR
         and3TotalR);
```

The monotonicity of the matching function is a longer proof but follows along the same lines.

$$\begin{array}{l}\vdash \forall ch, c, d.\ \ (\texttt{cle}\ c\ d\ \texttt{==}\ \texttt{true} \wedge \\ \qquad\qquad \texttt{typeOfChooser}\ ch\ \texttt{==}\ (\texttt{typeOfConst}\ c\texttt{,}\ \texttt{true}) \wedge \\ \qquad\qquad \texttt{typeOfChooser}\ ch\ \texttt{==}\ (\texttt{typeOfConst}\ d\texttt{,}\ \texttt{true})) \rightarrow \\ \qquad\qquad\qquad\qquad \texttt{le3}\ (\texttt{match}\ ch\ c)\ (\texttt{match}\ ch\ d)\ \texttt{==}\ \texttt{true}\end{array}$$

The function `le3` implements the ordering on three-valued booleans `uu`, `tt` and `ff`. Due to the presence of a second constant d, wherever the totality proof used a constant induction on c, it now includes an induction on d. As c and d have the same type, this extra induction does not introduce any extra subgoals (*cf.* `twoConstInduct` of Section 4.2.1.) For the `C` choosers this is proved by case analysis, and for the `B` and `T` choosers the proof relies on the monotonicity of `and3` and `or3`.

A nice corollary of the mononicity of `match` is the following:

$$\begin{array}{l}\vdash \forall c, d.\ \ \texttt{typeOfConst}\ c\ \texttt{==}\ \texttt{typeOfConst}\ d \wedge \texttt{cle}\ c\ d\ \texttt{==}\ \texttt{true} \rightarrow \\ \qquad\quad \texttt{NOT}\ (\exists ch.\ \ \texttt{typeOfChooser}\ ch\ \texttt{==}\ (\texttt{typeOfConst}\ c\texttt{,true}) \wedge \\ \qquad\qquad\qquad \texttt{match}\ ch\ c\ \texttt{==}\ \texttt{tt} \wedge \texttt{match}\ ch\ d\ \texttt{==}\ \texttt{ff})\end{array}$$

It states that there does not exist a chooser which can distinguish between two constants one of which is smaller than the other. In other words, increasing the definedness (in the data ordering sense, rather than partiality) of a constant cannot affect which branch of an `IF` statement we take.

4.3.2 Monotonicity of the Dynamic Semantics

The totality and monotonicity of the reduction function `reduce` are interdependent, and have to be proved simultaneously. The monotonicity of `reduce` in its first argument (the environment) may be proved independently of its monotonicity in its second argument (the circuit), but not *vice versa*. We will indicate in the proof where this fact is needed. Due to the mutual recursion of the `reduce` and `iterate` functions, any property which is to be proved about `reduce` will need a related invariant on `iterate`. It is this which makes the statement of the theorem which we will prove very large. The simplest monotonicity theorem we can prove is the following.

$$
\begin{array}{l}
\vdash\ \forall envl, envr, e, t.\\
\quad \texttt{lle}\ envl\ envr\ \texttt{==}\ \texttt{true}\ \wedge\\
\quad \texttt{map typeOfConst}\ envl\ \texttt{==}\ \texttt{map typeOfConst}\ envr\ \wedge\\
\quad \texttt{typeOfExpr (map typeOfConst}\ envl\texttt{)}\ e\ \texttt{==}\ \texttt{(}t\texttt{, true)}\ \rightarrow\\
\qquad\qquad \exists c1l, f1l, c1r, f1r.\ \ \begin{array}[t]{l}
\texttt{reduce}\ envl\ e\ \texttt{==}\ \texttt{(}c1l\texttt{,}\ f1l\texttt{)}\ \wedge\\
\texttt{reduce}\ envr\ e\ \texttt{==}\ \texttt{(}c1r\texttt{,}\ f1r\texttt{)}\ \wedge\\
\texttt{cle}\ c1l\ c1r\ \texttt{==}\ \texttt{true}\ \wedge\\
\texttt{typeOfConst}\ c1l\ \texttt{==}\ t\ \wedge\\
\texttt{typeOfConst}\ c1r\ \texttt{==}\ t
\end{array}
\end{array}
$$

For clarity we omit the invariant on `iterate` from the above statement. The theorem expresses that

For all environments *envl* and *envr*, and expressions *e*,

> if *envl* and *envr* are ordered, and equal as type environments, and
>
> > *e* is well-typed in the type environment
>
> then there exist two tuples (*c1l*,*f1l*) and (*c1r*,*f1r*) such that
>
> > `reduce` *envl e* and `reduce` *envr e* evaluate to these values respectively, and the output values *c1l* and *c1r* are ordered, and have the same type as the expression *e*.

The seemingly spurious observation that the result constants have the same type as the original expressions (or at least that the constants have the same type) is necessary, as we shall see in the section dealing with the `LET` statement. Note that the fact that the output tuples exist is a non-trivial observation. It means that the recursion of a well-typed expression in `iterate` always terminates. This is not necessarily the case for all expressions. The oscillating delayless feedback NOT gate `iterate [hi] (LetRec (hi, If (Var 0,lo,hi,C hi),Var 0))` does not terminate and is an example of a non-denoting term in LAMBDA. Although a logic of partial terms involves additional existence conditions, it does allow partial functions. In our case stating the `iterate` function as a partial function is more natural than forcing it to be total.

The above theorem may be expressed in the following diagrammatic form.

$$
\begin{tikzcd}
c1l,\ f1l & c1r,f1r \arrow[l, "\texttt{cle}"'] \\
envl \arrow[u, Rightarrow, "\texttt{reduce}\ envl\ e"] & envr \arrow[l, "\texttt{lle}"] \arrow[u, Rightarrow, "\texttt{reduce}\ envr\ e"']
\end{tikzcd}
$$

Figure 4–1: Monotonicity of `reduce` in Its First Argument.

Single arrows denote ordering, with the arrow pointing to the smaller object. The particular ordering function is used as a label for the arrow. In case of the lower horizontal arrow the value environments are ordered using the `lle` function, and for the top horizontal arrow the constants of the (constant, expression) pair are also ordered. Double arrows indicate dynamic semantics evaluation, with the arrow pointing towards the result.

However, this theorem is less useful than would appear at first sight. Consider the circuit `Delay (hi, Var 0)`. With *envl* `= [bit]` and *envr* `= [lo]` we get as outputs `(hi, Delay (bit,Var 0))` and `(hi, Delay (lo,Var 0))` as respective outputs. As we would expect, `cle hi hi` holds. But at the next time step we cannot use the theorem because we are dealing with two *distinct* circuits. This arises from the fact that the state is part of the circuit description, and different environments may give rise to different states. Thus we have to modify the theorem to deal with two expressions, which are identical apart from their internal states. The function `shapeEq`, defined on page 121, implements this notion. Thus we want to prove the following more involved result about `reduce`:

For all environments *envl* and *envr*, and expressions *e0l* and *e0r*,

> if *envl* and *envr* are ordered, and equal as type environments and
>
> > *e0l* and *e0r* are well-typed in the type environments, and
> > have the same shape and are ordered
>
> then there exist two tuples (*c1l*, *e1l*) and (*c1r*, *e1r*) such that
>
> > `reduce` *envl e0l* and `reduce` *envr e0r* evaluate to these values respectively, and the output values *c1l* and *c1r* are ordered, and have the same type as the original expression.
> > Moreover, the output expressions *e1l* and *e1r* are ordered and have the same shape, and
> > they have the same type and shape as the original expressions.

Shape equality of input and output expressions means that, for example, an adder circuit will not become a multiplier after some time, no matter what its input is. We do not allow dynamically changing circuit descriptions: only the state in delays and the approximations in the `LET REC` statements change. We abbreviate the conclusions of the theorem by THMR:

```
val THMR#(e0l, e0r, envl, envr, t) =
    ∃c1l, e1l, c1r, e1r.
      reduce envl e0l == (c1l, e1l) ∧
      reduce envr e0r == (c1r, e1r) ∧
      cle c1l c1r == true ∧          (* a *)
      ple e1l e1r == true ∧          (* b *)
      shapeEq e1l e1r == true ∧
      shapeEq e0l e1l == true ∧
      shapeEq e0r e1r == true ∧
      typeOfConst c1l == t ∧
      typeOfConst c1r == t ∧
      typeOfExpr (map typeOfConst envl) e1l == (t, true) ∧
      typeOfExpr (map typeOfConst envr) e1r == (t, true)
```

The labels **a** and **b** will be used later, in figure 4–4. We can now state the theorem in LAMBDA as follows.

```
⊢ ∀nl, nr, e0l, e0r, envl, envr, t.
    sizeOfExpr e0l == nl ∧
    sizeOfExpr e0r == nr ∧
    lle envl envr == true ∧
    ple e0l e0r == true ∧
    shapeEq e0l e0r == true ∧
    map typeOfConst envl == map typeOfConst envr ∧
    typeOfExpr (map typeOfConst envl) e0l == (t, true) ∧
    typeOfExpr (map typeOfConst envr) e0r == (t, true) →
                THMR#(e0l, e0r, envl, envr, t)
```

We will use the abbreviation THM#($nl, nr, e0l, e0r, envl, envr, t$) to denote the theorem without the leading universal quantifiers. The value environments *envl* and *envr* must have the same length by virtue of `map typeOfConst` *envl* `== map` `typeOfConst` *envr*. In principle the length of *envl* could be smaller than that of *envr*, as long as they are ordered element-wise on the initial segment of length `length` *envl* of *envr*. If both expressions do not access a particular variable (element in the environment) then its value and type are irrelevant. The condition that both environments are the same when viewed as type environments, ensures that the types of unused variables are the same. Note that if *e0l* and *e0r* have the same shape, and *e0l* is well-typed it does not follow that *e0r* is well-typed. (*e0r* could have non-bottom initial approximations in its recursive `LET` statements.) This may be seen from the definitions of `typeOfExpr` and `shapeEq` on pages 119 and 121 respectively.

In graphic form we obtain figure 4–2.

$$\begin{tikzcd}
c1l, e1l & & c1r, e1r \arrow[ll, "\texttt{cle, ple}"'] \\
\\
envl, e0l \arrow[uu, twoheadrightarrow, "\texttt{reduce}\ envl\ \ e0l"] & & envr, e0r \arrow[ll, "\texttt{lle, ple}"] \arrow[uu, twoheadrightarrow, "\texttt{reduce}\ envr\ \ e0r"']
\end{tikzcd}$$

Figure 4–2: Monotonicity of `reduce` in Both Arguments.

We have added a universal quantification over two natural numbers *nl* and *nr* which must be equal to the size of *e0l* and *e0r* respectively. This enables a more powerful proof on the size of expressions, instead of a structural induction on expressions. We will indicate in the proof where we need this extra power.

The proof proceeds by a nested general natural number induction on the *nl* and *nr*. This induction takes the following form, and is called `genInduct` in LAMBDA.

`E` n', $\forall m.\quad m$ `<` n' `== true` $\rightarrow$ `P#(`m`)` $\vdash$ `P#(`n'`)`
`--------------------`
`E` n $\vdash$ `P#(`n`)`

The reason for using this rather than the standard natural number induction is that the size of the subexpressions need not be exactly one less than the size of the total expression. Thus the standard inductive case `P#(`n`)` $\vdash$ `P#(S` n`)` is not suitable. A structural induction on the two expressions *e0l* and *e0r* is used after the general natural number induction. As we know that these two expressions have the same shape, this introduces only eight subgoals rather than sixty four subgoals. Recall that the shape equality is equivalent to comparing abstract syntax trees, in which only constants may be different. To illustrate the monotonicity proof we consider the three most interesting subgoals, the `IF`, the `LET`, and the recursive `LET` statement.

The If Statement

The `IF` statement is straightforward. The variables *e0l* and *e0r* are equal to `If(`*e0l*`,`*f0l*`,`*g0l*`,`*ch*`)` and `If(`*e0r*`,`*f0r*`,`*g0r*`,`*ch'*`)` in this case. Shape equality implies that *ch* and *ch'* are in fact equal. We instantiate the induction hypothesis for the three pairs of subexpressions of the `IF` statement; we use `THM#(`...`,`*e0l*`,`*e0r*`,`...`)`, *etc.* to obtain the conclusion of the theorem for the subevaluations. Recall that at the heart of the definition of the evaluation of the `IF` statement lies a match, which returns `uu`, `tt`, or `ff`.

```
fun reduce l (If (e1, e2, e3, ch)) = ...
    (case match ch c1 of
          uu => bottomOfConst c2 |
          tt => c2 |
          ff => c3,
     If (f1, f2, f3, ch)) ...
```

If *c0l* and *c0r* are the (constant) outputs from evaluating *e0l* and *e0r* respectively, `match` *ch* *c0l* and `match` *ch* *c0r* give a total of nine combinations of `tt`, `ff`, and `uu`. However, as the subexpressions' outputs are ordered (by the induction hypotheses) not all of the nine combinations are valid. Using the induction hypothesis for *e0l* and *e0r* we conclude that `cle` *c0l c0r*. Since `match` is monotone it follows that `le3 (match` *ch c0l*`) (match` *ch c0r*`)` holds. So the only valid combinations of `uu`, `tt` and `ff` are `(uu,uu)`, `(uu,tt)`, `(uu,ff)`, `(tt,tt)` and `(ff,ff)`. The last but one of these indicates that both `IF` statements choose the `THEN` branch, which we know to be ordered by virtue of the induction hypothesis for *f0l* and *f0r*. In the final case both select the `ELSE` branch, for which monotonicity also holds by the induction hypothesis for *g0l* and *g0r*. In the first three cases the first `IF` statement delivers the bottom value because it cannot decide which of its two outputs it should deliver. (This is what the `uu` value from the matching process indicates.) We know than the bottom value is less than or equal than any other value of its type. In particular the value from the first `IF` is less than or equal to the output of the second `IF`.

The Let Statement

The `LET` follows the same approach as the `IF` statement. If `Let (`*e0l*`,` *f0l*`)` is the first expression, and `Let (`*e0r*`,` *f0r*`)` the second, we know that they are both well-typed, and have the same shape and type *t*. When we consider how the type of a `Let` is computed it is not obvious, however, that the types of the subexpressions *e0l* and *e0r* are also equal.

```
fun typeOfExpr te (Let (e1, e2)) =
    (fn (t1, b1) =>
    (fn (t2, b2) =>
            (t2, b1 && b2))
            (typeOfExpr (t1::te) e2)) (typeOfExpr te e1) | ...
```

For all we know, the intermediate types *t1* could diverge, but the types *t2* of the whole `Let` constructs could be the same. If this were so, we could not apply the induction hypothesis because it assumes that both types are the same *t*. As it happens, we can prove that if two expressions are both well-typed, and are shape equal then their types are the same.

$\vdash \forall e0l, f0l, l, t1, t2.$ `shapeEq` $e0l$ $f0l$ `== true` $\wedge$
`typeOfExpr` l $e0l$ `== (`$t1$`, true)` $\wedge$
`typeOfExpr` l $f0l$ `== (`$t2$`, true)` $\rightarrow$ $t1$ `==` $t2$

We can not prove the stronger result which states that if two expressions are shape equal, and one of the expressions is well-typed then the other one is well-typed with the same type. This is solely a consequence of `typeOfExpr`'s insistence on bottom initial approximations for recursive `LET` constructs, which shape equality does not preserve. At the start of this section (on page 135) we mentioned that the following conclusions are essential: (1) `typeOfConst` $c1l$ `==` t and (2) `typeOfConst` $c1r$ `==` t, where t is the type of the defining expressions $e0l$ and $e0r$. It is at this point that we need them.

When we instantiate the induction hypothesis for $f0l$ and $f0r$ with value environments $c1l$::$envl$ and $c1r$::$envr$ respectively, we have to prove that the value environments are the same when viewed as type environments. That is, `map typeOfConst (`$c1l$::$envl$`) == map typeOfConst (`$c1r$::$envr$`)`. It follows from the lemma above that the types of $e0l$ and $e0r$ are the same, and thus, using (1) and (2) above, that `typeOfConst` $c1l$ `== typeOfConst` $c1r$. The `LET` subgoal may be proved without any further complications. The lemma must also be used in the subgoals dealing with the indexing operators, where we don't care about the type of the part the indexing throws away.

A First Attempt at The LetRec Statement

In all subgoals but the recursive `LET` we are not interested in the invariant on the `iterate` function. Moreover, we could have proved all these subgoals using a simple nested structural induction on expressions. Only in the `LET REC` do we need the natural number induction on the size of expressions, which is more general than the structural induction. Let us consider what happens if we try to apply the simple approach which we used so far to the `LetRec`. Let $e0l$ in the theorem be `LetRec(`$c0l$`, `$e0l$`, `$f0l$`)` in this subgoal and $e0r$ be `LetRec(`$c0r$`, `$e0r$`, `$f0r$`)`. We have to prove the following goal using the given induction hypotheses, ordered in decreasing generality.

```
1:  ∀m.  m < nl == true → ∀nr,e,f,t.  THM#(m,nr,e,f,t)
2:  ∀m.  m < nr == true → ∀e,f,t.  THM#(nl,m,e,f,t)
3:  ∀f,t.  THM#(nl,nr,e0l,f,t)
4:  ∀t.  THM#(nl,nr,e0l,e0r,t)
5:  nl == sizeOfExpr (LetRec (c0l,e0l,f0l))
6:  nr == sizeOfExpr (LetRec (c0r,e0r,f0r))
... ⊢ THMR#(nl,nr,LetRec(c0l,e0l,f0l),LetRec(c0r,e0r,f0r),t)
```

We have omitted the existence hypotheses, and the antecedent of the implication in THM, of which the fifth and sixth hypotheses are parts. We have to evaluate the following term in THMR.

```
reduce envl (LetRec (c0l,e0l,f0l))
```

We expand it, using the definition of `reduce`, and arrive at a subterm

```
iterate envl e0l c0l
```

which, according to `iterate`'s definition, is equal to

```
(fn (d,f) => if ceq c0l d then (d,f) else ...) (reduce (c0l::envl))
```

Using one of the induction hypotheses above, we obtain an answer (*c1l*,*e1l*) for `reduce` (*c0l*::*envl*) *e0l*. Substituting this into the previous expression gives

```
if ceq c0l c1l then (c1l,e1l) else iterate envl e0l c1l
```

We have computed the second approximation *c1l*, using the first approximation *c0l*. Now `iterate` checks if we have reached a fixed point (*i.e.* *c0l* and *c1l* are equal.) If we have, we return the result (*c1l*,*e1l*). If, on the other hand, *c0l* and *c1l* are distinct we have not reached a fixed point, and we call `iterate` again with the most recent approximation *c1l*. We get a similar situation for the right hand side expression *e0r*. Thus we have four possibilities: (1) both recursions have arrived at their fixed points, that is, `ceq` *c0l* *c1l* and `ceq` *c0r* *c1r*; or (2) the first recursion has fixed, but not the second (`ceq` $c0r$ $c1r$ `== false`); or (3) the first recursion must iterate again, but the second fixes; or (4) both recursions must iterate again. These cases are obtained by doing a boolean induction (case analysis) on the value of `ceq` $c0l$ $c1l$ and `ceq` $c0r$ $c1r$. (Note that this is not the same as doing an induction on truth values, which is not possible — see Section 4.1.) The first case is trivial: it can be discharged after some rewriting. In the remaining cases we can compute the next approximations $c2l$ and $c2r$. If we do this, however, we end up having to decide whether we have arrived at a fixed point this time. If we have, we can stop, otherwise we have to iterate again. We can continue this process *ad infinitum*. The problem is that we can compute every next approximation, but we have no guarantee of a finite number of iterations.

To ensure this we define an invariant on `iterate`: `THMI#(`$nl, nr, c0l, c0r, envl, envr, e0l, e0r, c1l, c1r, e1l, e1r, t$`)` which will be defined on page 147. First, we will try to give a more intuitive account of the definition in the following figure.

$$\begin{tikzcd}
cnl, enl & cnr, enr \arrow[l, "\texttt{cle, ple}"'] \\
c1l, e1l \arrow[u, Rightarrow, "*"', "\texttt{iterate}\ envl\ e0l\ c1l"] & c1r, e1r \arrow[l, "\texttt{cle, ple}"'] \arrow[u, Rightarrow, "*", "\texttt{iterate}\ envr\ e0r\ c1r"'] \\
(c0l{::}envl), e0l \arrow[u, Rightarrow, "\texttt{reduce}\ (c0l{::}envl)\ e0l"] & (c0r{::}envr), e0r \arrow[l, "\texttt{lle, ple}"'] \arrow[u, Rightarrow, "\texttt{reduce}\ (c0r{::}envr)\ e0r"']
\end{tikzcd}$$

Figure 4–3: The `THMI` Invariant on `iterate`.

A star on the double arrow indicates that `reduce` is applied zero or more times to arrive at the answer. (This is `iterate`'s purpose.) The `THMI` invariant introduces some orderings between the different approximations, *i.e.* vertical arrows in the diagram. We have omitted these arrows here, and have them shown separately in the following figure 4–4. The values $c0l$ and $c0r$ are the first approximations, $c1l$ and $c1r$ the second, *etc.* In general we have `reduce` (cil`::`$envl$) $e0l$ `==` ($c(i{+}1)l$, $e(i{+}1)l$) and `reduce` (cir`::`$envr$) $e0r$ `==` ($c(i{+}1)r$, $e(i{+}1)r$). Note that we use $e0l$ and $e0r$ every time we compute a new iteration; this was explained in Section 3.3.3. cnl and cnr are the fixed points. All of the cil are equal for $i \geq n$.

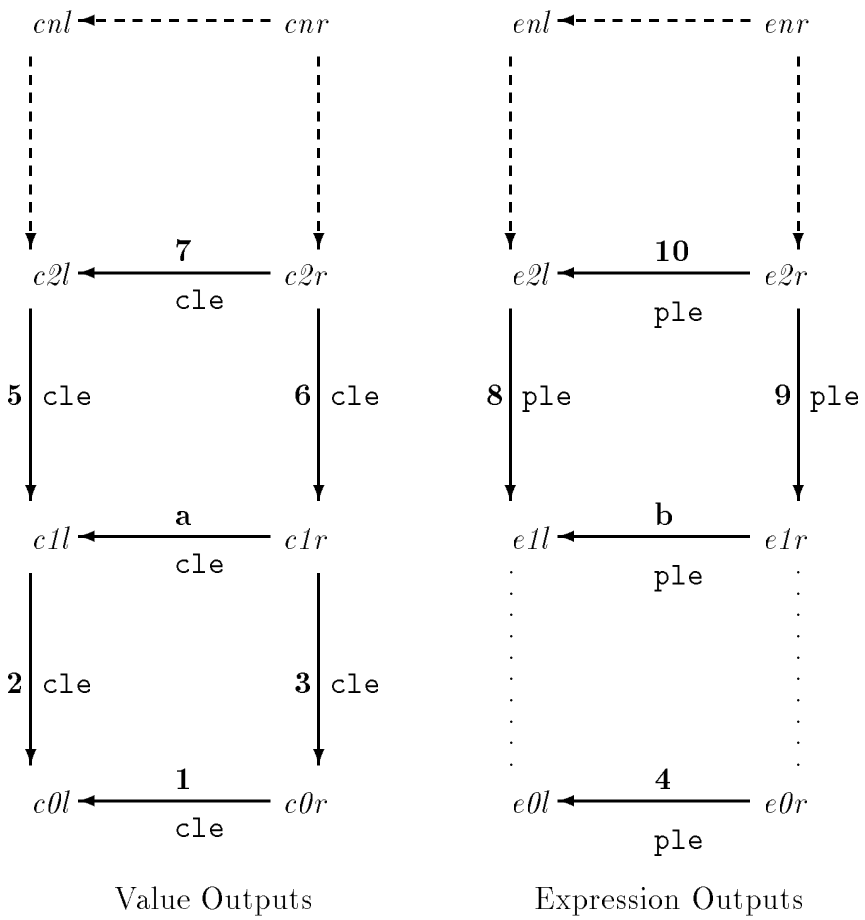

Figure 4–4: Ordering Approximations During a Fixed Point Computation.

The numbering of the arrows (**1** to **10**) corresponds to the numbering of the orderings in the definition of `THMI` on page 147, and arrows **a** and **b** refer to orderings in `THMR` on page 137. The dashed lines indicate that the lattice continues to the fixed points, after which the values remain the same. The dotted lines in the right hand diagram indicates that the ordering does *not* hold. We need two squares (and therefore variables *c2l*, *c2r*, *e2l*, and *e2r*) because `ple` *e0l e1l* and `ple` *e0r e1r* do not hold. The reason for this is that the passage of time happens at this point: *e0l* is the circuit at time t and *e1l* is the circuit at time $t+1$. In general there is no ordering relation between these two circuit descriptions. The delayed NOT gate which is fed back into itself illustrates this:

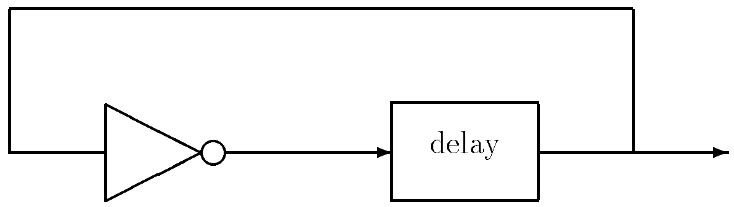

Figure 4–5: A Delayed Feedback NOT Gate.

The circuit at time zero is:

```
LetRec (bit, Delay (hi, If (Var 0, lo, hi, C hi)), Var 0)
```

We have the following computations in the iterations (within one time step):

`reduce (bit::`*envl*`) (Delay (hi,...)) == (hi,Delay (lo,...))`

`reduce (hi::`*envl*`) (Delay (lo,...)) == (hi,Delay (lo,...))`

`reduce (hi::`*envl*`) (Delay (lo,...)) == (hi,Delay (lo,...))`

The delay will contain `hi` at times $2n$, and `lo` at times $2n+1$. So at time $2n$ *e0l* is `Delay (hi,...)` and *e1l* will be `Delay (lo,...)`. The expression *e2l* will also be `Delay (lo,...)`, so that `ple` *e1l e2l* holds. However, `ple` *e0l e1l* does not hold at any time.

Given this informal explanation the invariant `THMI` is defined on page 147. Although the statement is very large, all it says is that `iterate` is monotone in its inputs (value environment, expressions and approximations), and preserves the shape of expressions. It also preserves the types of expressions and constants. These conditions 'horizontally' relate the inputs with inputs (**1** and **4**), and output with outputs (**7** and **10**.) Three additional invariants 'vertically' link inputs with outputs: (i) the output of the iteration (*i.e.* the next approximation) is more defined than the input approximation (**2**, **3**, **5** and **6**, *e.g.* `cle` *c1l c2l*), and (ii) the output expression of the second approximation is greater than the output expression of the first approximation (**8** and **9**, *e.g.* `ple` *e1l e2l*), and (iii) the output expression of the second approximation has the same shape as the output expression of the first approximation (*e.g.* `shapeEq` *e1l e2l*.) The orderings numbered **1** to **10** correspond to the numbered arrows in figure 4–4.

An 'arrow chasing' interpretation of `THMI`, with reference to diagram 4–4, may be given as follows. If we have the four arrows **1** to **4** (these form the shape of a 'U' in the diagram) we can infer arrows **5** to **10**, which form the shape of two upside down 'U's in the diagram. Thus `THMI` is used to obtain the 'vertical' ordering relations **5**, **6**, **8** and **9**, and the 'horizontal' relations **7** and **10**. The hypothesis `THMR`, on the other hand, is used to infer the intermediate 'horizontal' orderings **a** and **b** in the diagram. From **1** and **4** we can conclude **a** and **b**. In the first attempt to prove the monotonicity of the `LET REC` statement we used only horizontal arrows **1**, **a**, **7**, ... and **4**, **b**, **10**, ... Using `THMI` we obtain extra vertical information, consisting of arrows **2**, **3**, **5**, **6**, **8**, and **9**.

The definition of `THMI` is given overleaf.

```
val THMI#(n0l,n0r,c0l,c0r,envl,envr,e0l,e0r,c1l,c1r,e1l,e1r,t) =
    sizeOfConst c0l == n0l ∧
    sizeOfConst c0r == n0r ∧
    typeOfConst c0l == t ∧
    typeOfConst c0r == t ∧
    typeOfExpr (map typeOfConst (c0l::envl)) e0l == (t, true) ∧
    typeOfExpr (map typeOfConst (c0r::envr)) e0r == (t, true) ∧
    cle c0l c0r == true ∧          (* 1 *)
    cle c0l c1l == true ∧          (* 2 *)
    cle c0r c1r == true ∧          (* 3 *)
    ple e0l e0r == true ∧          (* 4 *)
    shapeEq e0l e0r == true ∧
    reduce (c0l::envl) e0l == (c1l,e1l) ∧
    reduce (c0r::envr) e0r == (c1r,e1r) →
    ∃c2l,e2l,c2r,e2r.
     iterate envl e0l c1l == (c2l,e2l) ∧
     iterate envr e0r c1r == (c2r,e2r) ∧
     cle c2l c2r == true ∧         (* 7 *)
     cle c1l c2l == true ∧         (* 5 *)
     cle c1r c2r == true ∧         (* 6 *)
     ple e2l e2r == true ∧         (* 10 *)
     ple e1l e2l == true ∧         (* 8 *)
     ple e1r e2r == true ∧         (* 9 *)
     shapeEq e2l e2r == true ∧
     shapeEq e1l e2l == true ∧
     shapeEq e1r e2r == true ∧
     typeOfConst c2l == typeOfConst c1l ∧
     typeOfConst c2r == typeOfConst c1r ∧
     typeOfExpr (map typeOfConst (c2l::envl)) e2l == (t,true) ∧
     typeOfExpr (map typeOfConst (c2r::envr)) e2r == (t,true);
```

We now modify the statement of the overall theorem to include the `THMI` invariant. We will prove $\forall nl, nr, e0l, e0r, envl, envr, t.$ `THM#`$(nl, nr, e0l, e0r, envl, envr, t)$.

```
val THM#(nl,nr,e0l,e0r,envl,envr,t) =
    sizeOfExpr e0l == nl ∧
    sizeOfExpr e0r == nr ∧
    lle envl envr == true ∧
    ple e0l e0r == true ∧
    shapeEq e0l e0r == true ∧
    map typeOfConst envl == map typeOfConst envr ∧
    typeOfExpr (map typeOfConst envl) e0l == (t, true) ∧
    typeOfExpr (map typeOfConst envr) e0r == (t, true) →
               (THMR#(e0l,e0r,envl,envr,t) ∧
                ∀xl,yl,zl,xr,yr,zr.
                  e0l == LetRec (xl,yl,zl) ∧
                  e0r == LetRec (xr,yr,zr) →
                        ∀n0l,n0r,c0l,c0r,c1l,c1r,e1l,e1r.
                          THMI#(n0l,n0r,c0l,c0r,envl,envr,yl,yr,
                                c1l,c1r,e1l,e1r,typeOfConst xl));
```

Recall that `THMR` was defined on page 137. This new statement includes `THMI`. We have not simply changed the conclusion to `THMR#(...)` $\wedge$ `THMI#(...)` because this would entail proving `THMI` for all expression constructors. Although in theory `THMI` could make sense for non-`LET REC` constructors we do not want to do unnecessary work. Note that in a structural induction on the $e0l$ and $e0r$ expressions we cannot use the `THMI` part of the hypotheses. The reason for this is that we do not know whether the subexpressions are `LetRec` statements or not. We therefore use `THMI` only when $e0l$ and $e0r$ are both `LetRec` statements. Before we do this, however, we take a look at some subproofs we will use during the proof of the `LET REC` statement.

How To Prove Orderings Of Approximations

It is helpful to step back at this point and take a more abstract view of what we are trying to prove. It may be helpful to refer to figure 4–4, and the definition of `THMI` on page 147 in the following. We have two `LET REC` statements which are computing their respective least fixed points. We have to prove that the fixed point on the left hand side is less than or equal to the fixed point on the right hand side. Both sides start with a bottom approximation; thus *c0l* `=` *c0r* `=` bottom. Every application of `reduce` to `(`*cil*`::`*envl*`)` *e0l* and `(`*cir*`::`*envr*`)` *e0r* amounts to computing the next approximation *c(i+1)l* and *c(i+1)r* respectively. This corresponds to computing the next horizontal arrows **a** and **b** and a `THM` induction hypothesis is used for this. We shall call this a 'horizontal instantiation' of the induction hypothesis. As we shall see, we also need the vertical arrows `cle` *cil* *c(i+1)l* and `cle` *cir* *c(i+1)r*. For expressions we need all vertical arrows, except `ple` *e0l* *e1l* and `ple` *e0r* *e1r*, as explained previously. We now describe how to prove the two vertical orderings.

If we know that `cle` *cil* *c(i+1)l* holds then we can prove `cle` *c(i+1)l* *c(i+2)l* as follows. We use the most general `THM` induction hypothesis:

$\forall m.$ m `<` nl `== true` $\rightarrow \forall nr, e, f, t.$ `THM#(`m, nr, e, f, t`)`

and instantiate *m* with `sizeOfExpr` *e0l*, which we can easily prove to be less than *nl* `= sizeOfExpr (LetRec (`*c0l*`,`*e0l*`,`*f0l*`))`.[3] We instantiate *nr* with *nl* also, and both *e* and *f* with *e0l*. Thus we pretend that the left vertical arrow (`cle` *cil* *c(i+1)l*) (arrow **2** in 4–4) is the horizontal arrow (arrow **1**.) We can easily discharge the antecedent of the hypothesis, and obtain the ordering `cle` *c(i+1)l* *c(i+2)l*. In diagrammatic form this corresponds to the left diagram in figure 4–6. The diagram on the right hand side shows the effect in the original lattice.

[3] See the discussion on the next page about the inadequacy of structural induction at this point.

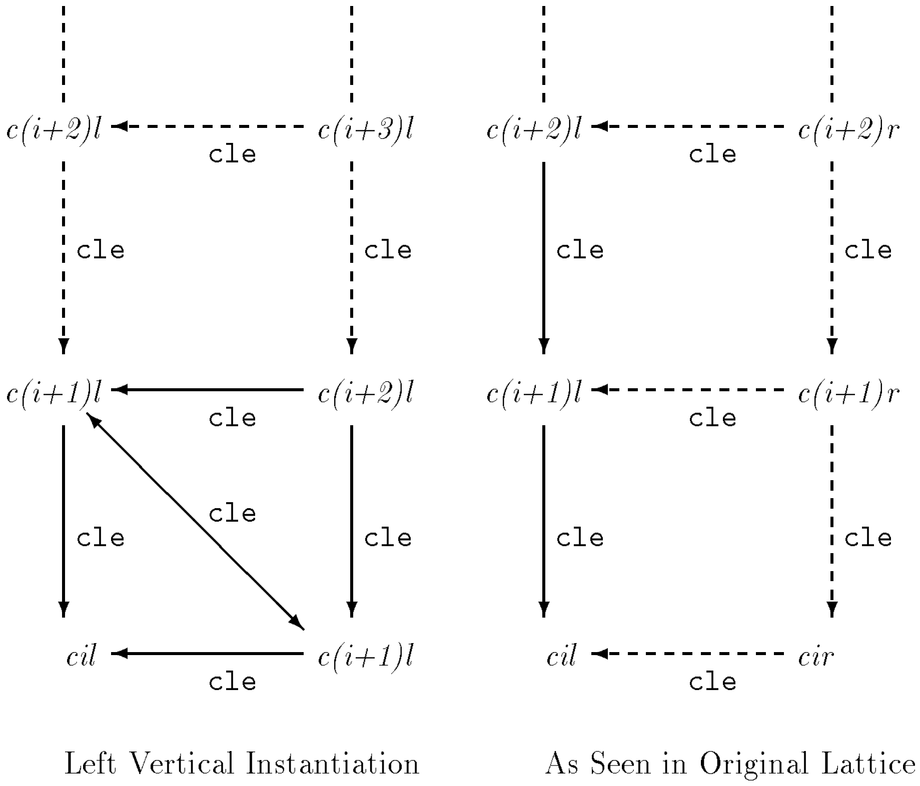

Figure 4–6: 'Left Vertical' Instantiation of Induction Hypothesis

The double arrow indicates equality; by identifying the two values at the ends we see that we have proved the ordering `cle` *c(i+1)l c(i+2)l*.

We can do a similar trick on the right hand side. Assuming `cle` *cir c(i+1)r* holds we can prove `cle` *c(i+1)r c(i+2)r*. In this case we have to instantiate m in the induction hypothesis with `sizeOfExpr` *e0r*, and have to do slightly more work to prove that it is less than *nl*, where *nl* is equal to `sizeOfExpr (LetRec (`*c0l*`,`*e0l*`,`*f0l*`))`.

For the vertical instantiations we need the general natural number induction on *nl* and *nr* (page 138.) It would not be enough to use the hypotheses from the structural induction on expressions. The structural induction hypotheses 3 and 4 on page 142 have *e0l* and *e0r* fixed as the circuit on the left and the circuit on the right. This prevents us doing the 'vertical instantiations' because they depend on using the same expressions (but different value environments) for the instantiation of the hypotheses. Moreover, for both vertical instantiations we rely on the fine comparison `sizeOfExpr` performs. Structural induction on expressions ignores all subexpressions of a `LetRec` which are not expressions. In other words, we could not simply have had something along the lines of $\forall e, f.$ `THM#(`e`,`f`)` because the structural induction wouldn't have known the difference between `LetRec (`$cil, e0l, f0l$`)` and `LetRec (`$c(i{+}1)l, e0l, f0l$`)`, which is crucial for the vertical instantiations.

So, using 'horizontal', 'left vertical' and 'right vertical' instantiations of the `THM` induction hypothesis we can find our way around the lattice of orderings, displayed in figure 4–4.

A Note on the Induction Principle

The subgoal in which both $e0l$ and $e0r$ are `LetRec` constructors has the following form.

```
... ⊢ THMR#(nl, nr,LetRec(c0l, e0l,f0l),LetRec(c0r, e0r,f0r),t) ∧
      ∀xl, yl, zl, xr, yr, zr.
        LetRec (c0l, e0l,f0l) == LetRec (xl, yl, zl) ∧
        LetRec (c0r, e0r,f0r) == LetRec (xr, yr, zr) →
               ∀n0l, n0r, c0l, c0r, c1l, c1r, e1l, e1r.
                THMI#(n0l, n0r, c0l, c0r, envl, envr, yl, yr,
                      c1l, c1r, e1l, e1r,typeOfConst xl)
```

We have omitted a number of existence and induction hypotheses. The antecedent of the implication of `THM` has also been omitted. We can now apply the rule `and2R`, which allows us to use `THMI` in the proof of `THMR`.

```
⊢ Q
Q ⊢ P
--------------------
⊢ P ∧ Q
```

After rewriting the `LetRec ... == LetRec ...` terms and unifying the variables ($c0l$ with xl *etc.*), we obtain the following two subgoals.

```
***** Premise 2 *****
... ⊢ ∀n0l,n0r,c0l,c0r,c1l,c1r,e1l,e1r.
THMI#(n0l,n0r,c0l,c0r,envl,envr,e0l,e0r,c1l,c1r,e1l,e1r,typeOfConst c0l)
***** Premise 1 *****
∀n0l,n0r,c0l,c0r,c1l,c1r,e1l,e1r.
THMI#(n0l,n0r,c0l,c0r,envl,envr,e0l,e0r,c1l,c1r,e1l,e1r,typeOfConst c0l)
... ⊢ THMR#(nl,nr,LetRec(c0l,e0l,f0l),LetRec(c0r,e0r,f0r),t)
```

Thus we have to prove (1) that `THMR` holds for the `LetRec` case, assuming the invariant `THMI` on `iterate`, and (2) that `THMI` does really hold for `iterate`.

The proof of the monotonicity and totality of `reduce` depends on the same property for `iterate`. Premise 2 corresponds to the latter, and it is used as a hypothesis in premise 1, which corresponds to the former. It is tempting to think of premise 1 as a base case for some induction, but this is not correct. `THMI` is instantiated directly in the `THMR` case, because `THMI` is not an induction hypothesis.

The proof of the second premise `THMI` proceeds by a general natural number induction (`genInduct` on page 138.) The size *n0l* of the current approximation *c0l* may be regarded as the maximum height of the derivation tree of the semantics. That is, there can be at most `sizeOfConst` *c0l* iterations using *c0l* as the initial approximation. As `cle` x y implies `sizeOfConst` y `<=` `sizeOfConst` x, it follows that the bottom value can have the largest derivation tree, and a fully defined value a smallest derivation tree. The induction rule states that, assuming that the `THMI` holds for all smaller derivation trees, we have to prove that it holds for the current derivation tree. In other words, assuming `iterate` obeys the appropriate properties for smaller size `sizeOfExpr` *c(i+1)l*, they also hold for the current size `sizeOfExpr` *cil*. Of course, we don't think about sizes of approximations, we think about the approximations themselves. The statement then becomes more intuitive; assuming `iterate` obeys the appropriate properties for the next approximation *c(i+1)l* (corresponding to the recursive call in `iterate`; see the next page), we have to prove that they also hold for the current approximation *cil*. By using `genInduct` instead of the standard natural number induction rule `natInduct`, we don't have to prove a base case. The term `P#(0)` would correspond to a fully defined approximation, but the least fixed point is not necessarily fully defined. Moreover, the inductive step `P#(`n`)` $\vdash$ `P#(S` n`)` restricts us to increasing the (size of the) current approximation one step at a time, whereas the fixed point computation could omit intermediate steps.

The proof resembles a fixed point induction because the `THMR` case deals with bottom values. It is different, however, because premise 1 is not a base case for an induction. Moreover, in the inductive step of the proof of premise 2 we do not use the hypothesis for a less defined value (as is the case in a fixed point induction [118]), but instantiate it with a more defined value. A more defined value has a smaller size, corresponding to a smaller derivation tree.

After the discussions of the induction principle and the various 'horizontal' and 'vertical' induction hypothesis instantiations we now return to the proof.

The Monotonicity of The LetRec Statement

We will first describe the proof of the `THMR` premise. In this subproof, *c0l* and *c0r* are equal to the bottom values, because the well-typedness of the `LetRec` constructor ensures that the initial approximation is equal to the bottom value (see page 119.) As in our first attempt at the proof on page 141, we use a horizontal instantiation to obtain the answers (*c1l*, *e1l*) and (*c1r*, *e1r*) for `reduce (`*c0l*`::`*envl*`)` *e0l* and `reduce (`*c0r*`::`*envr*`)` *e0r* respectively. As before, this allows us to rewrite `iterate` *envl e0l c0l* to

`if ceq` *c0l c1l* `then (`*c1l*, *e1l*`) else iterate` *envl e0l c1l*

and a similar expression for `iterate` *envr e0r c0r*. In this version of the proof we use the `THMI` subgoal, to prove that the `iterate` expressions exist. The `iterate` expressions correspond to the second and further approximations.

$\forall$*n0l*, *n0r*, *c0l*, *c0r*, *c1l*, *c1r*, *e1l*, *e1r*.
`THMI#(`*n0l*, *n0r*, *c0l*, *c0r*, *envl*, *envr*, *e0l*, *e0r*, *c1l*, *c1r*, *e1l*, *e1r*,`typeOfConst` *c0l*`)`

Recall from figure 4–4 that we need arrows **1** to **4** to be able to use the hypothesis. **1** (`cle` *c0l* *c0r*) and **4** (`ple` *e0l* *e0r*) are given as part of the antecedent of the `THMR`; **2** (`cle` *c0l* *c1l*) and **3** (`cle` *c0r* *c1r*) are trivial to prove, because *c0l* and *c0r* are equal to the bottom value. We therefore instantiate the variables *n0l*, *n0r*, *c0l*, *c0r*, *c1l*, *c1r*, *e1l*, and *e1r* with `sizeOfConst` *c0l*, `sizeOfConst` *c0r*, *c0l*, *c0r*, *c1l*, *c1r*, *e1l*, and *e1r* respectively. We discharge the left hand side of `THMI`'s implication. This gives us fixed points *c2l* and *c2r*. We now perform a boolean case analysis on the values of the comparisons `ceq` *c0l* *c1l* and `ceq` *c0r* *c1r* in the expanded `iterate` term. This gives four cases, of which we can discharge all but the third subgoal easily. These cases are:

Case 1: `ceq` *c0l* *c1l* and `ceq` *c0r* *c1r*. That is, both the left and right hand side of the lattice have reached their fixed points *c0l* `==` *c1l* `==` *c2l* and *c0r* `==` *c1r* `==` *c2r*. Following this, we have to compute the output for the `LetRec` `(`*c0l*`,`*e0l*`,`*f0l*`)`. If *cnl* is the fixed point, this may be done by instantiating a `THMR` induction hypothesis with *cnl*`::`*envl* and *f0l*, *etc.* This proves this case for the `LetRec`.

Case 2: `ceq` *c0l* *c1l* but `ceq` *c0r* *c1r* `== false`. The left hand side fixes, but the right hand side has not reached its fixed point yet. This case proceeds as 1 above. The fact that the right hand side has not reached a fixed point does not matter, because we have already instantiated `THMI` so that we know that the fixed points will be *c2l* and *c2r*. Conceptually it might have been clearer to instantiate `THMI` after the case analysis, but this would have meant duplicating the instantiation effort.

Case 3: `ceq` *c0l* *c1l* `== false` and `ceq` *c0r* *c1r* `== true`. The left hand side iterates and the right hand side fixes. This case cannot occur, and we derive a contradiction from the hypotheses as follows. We know that `cle` *c0l* *c1l*, `cle` *c0r* *c1r* (this follows from `ceq` *c0r* *c1r* and the reflexivity of `cle`), `cle` *c0l* *c0r*, `cle` *c1l* *c1r*. These correspond to arrows **1**, **2**, **3** and **a** in figure 4–4. In fact, *c1r* is equal to *c0r*. Moreover, anything less than bottom must be equal to bottom, so that *c1l* is equal to bottom. This contradicts with `ceq` *c0l c1l* `== false`.

Case 4: `ceq` *c0l c1l* `== false` and `ceq` *c0r c1r* `== false`. Both sides have not arrived at their fixed points yet. This is proved as case 1.

This concludes the proof of the `THMR` case of the `LET REC`.

We will now show how to prove the invariant `THMI` on `iterate` (premise 2 on page 151.)

$$\ldots \vdash \forall n0l, n0r, c0l, c0r, c1l, c1r, e1l, e1r.$$
$$\texttt{THMI\#}(n0l, n0r, c0l, c0r, envl, envr, e0l, e0r, c1l, c1r, e1l, e1r, \texttt{typeOfConst}\ c0l)$$

We do a general natural number induction on both *n0l* and *n0r*. This corresponds to an induction on the size of the constants *c0l* and *c0r* because all other subexpressions of the `LetRec` constructors remain unchanged. It may also be regarded as an induction on the height of the derivation tree, because the size of the approximation indicates the maximum number of iterations which may be computed using the approximation. This gives the following total of induction hypotheses:

$$\begin{array}{l}
\texttt{1:}\quad \forall m.\quad m \texttt{ < } nl \texttt{ == true} \rightarrow \forall nr, e, f, t.\quad \texttt{THM\#}(m, nr, e, f, t) \\
\texttt{2:}\quad \forall m.\quad m \texttt{ < } nr \texttt{ == true} \rightarrow \forall e, f, t.\quad \texttt{THM\#}(nl, m, e, f, t) \\
\texttt{3:}\quad \forall f, t.\quad \texttt{THM\#}(nl, nr, e0l, f, t) \\
\texttt{4:}\quad \forall t.\quad \texttt{THM\#}(nl, nr, e0l, e0r, t) \\
\texttt{5:}\quad \forall m.\quad m \texttt{ < } n0l \texttt{ == true} \rightarrow \forall n0r, c0l, c0r, c1l, c1r, e1l, e1r. \\
\texttt{THMI\#}(m, n0r, c0l, c0r, envl, envr, e0l, e0r, c1l, c1r, e1l, e1r, \texttt{typeOfConst}\ c0l) \\
\texttt{6:}\quad \forall m.\quad m \texttt{ < } n0r \texttt{ == true} \rightarrow \forall c0l, c0r, c1l, c1r, e1l, e1r. \\
\texttt{THMI\#}(n0l, m, c0l, c0r, envl, envr, e0l, e0r, c1l, c1r, e1l, e1r, \texttt{typeOfConst}\ c0l) \\
\texttt{7:}\quad nl \texttt{ == sizeOfExpr (LetRec (}c0l, e0l, f0l\texttt{))} \\
\texttt{8:}\quad nr \texttt{ == sizeOfExpr (LetRec (}c0r, e0r, f0r\texttt{))} \\
\texttt{9:}\quad n0l \texttt{ == sizeOfConst } c0l \\
\texttt{10:}\quad n0r \texttt{ == sizeOfConst } c0r \\
\vdash \texttt{THMI\#}(n0l, n0r, c0l, c0r, envl, envr, e0l, e0r, c1l, c1r, e1l, e1r, \texttt{typeOfConst}\ c0l)
\end{array}$$

As usual we have omitted existence conditions and most parts of the antecedents of `THM` and `THMI`. Hypotheses 5, 6, 9, and 10 are new. Note that `THMI` rather than `THMR` is to be proved. Where *c0l* and *c0r* were bottom values previously, they may now be anything, although `cle` *c0l* *c0r* still holds of course. The antecedent of `THMI` on the right hand side gives us arrows **1** and **4** of figure 4–4. We use these in a horizontal instantiation of `THMR` to obtain arrows **a** and **b**. This corresponds to the outputs of the `reduce` (*c0l*`::`*envl*) *e0l* and `reduce` (*c0r*`::`*envr*) *e0r*. These are (*c1l*, *e1l*) and (*c1r*, *e1r*) respectively. In the `THMR` case we could just instantiate the `THMI` hypothesis 5 to obtain the fixed points *c2l* and *c2r*, because it was not an inductive hypothesis. Here, however, it is shielded by the induction condition *m* `<` *n0l*. So we have to do the boolean case analysis on `ceq` *c0l* *c1l* and `ceq` *c0r* *c1r* in the expanded `iterate` term first. This gives us four cases, which are more involved than those of the `THMR` case. These cases are:

Case 1: `ceq` *c0l* *c1l* and `ceq` *c0r* *c1r*. That is, both the left and right hand side of the lattice have reached their fixed points. This subgoal is proved by straightforward rewriting.

Case 2: `ceq` *c0l* *c1l* but `ceq` *c0r* *c1r* `== false`. The left hand side fixes, but the right hand side has not reached its fixed point yet. We use a right vertical instantiation of `THM` hypothesis 1, which gives `cle` *c1r* *c2r* (arrow **6** of figure 4–4.) As *c1l* is equal to *c0l* we obtain `cle` *c0l* *c1r* by transitivity. The ordering `>=` on sizes of constants strictly reflects `cle`. This is stated as follows in LAMBDA:

$\vdash \forall x, y.$ `cle` x y `== true` $\rightarrow$ `sizeOfConst` y `<=` `sizeOfConst` x

From `cle` *c0r* *c1r* we can therefore conclude that `sizeOfConst` *c1r* `<=` `sizeOfConst` *c0r* holds. This, in conjunction with the inequality of *c0r* and *c1r*, implies `sizeOfConst` *c1r* `<` `sizeOfConst` *c0r*. This term is equal to `sizeOfConst` *c1r* `<` *n0r*, and this can be used to obtain induction hypothesis 6. We use the least general of the two `THMI` induction hypotheses because the value on the left hand side ($c0l$) does not change, as hypothesis 5 requires. Thus, we instantiate m with `sizeOfConst` $c1r$, discharge the induction condition, and instantiate $c0l$, $c0r$, $c1l$, $c1r$, $e1l$, $e1r$ with $c0l$, $c0r$, $c0l$, $c1r$, $e1l$, $e1r$ respectively. After discharging the antecedent of `THMI`, we conclude that `iterate` $envl\ e0l\ c1l$ `==` $(c2l, e2l)$, and `iterate` $envr\ e0r\ c2r$ `==` $(c3r, e3r)$. The former is equal to `iterate` $envl\ e0l\ c0l$ `==` $(c0l, e1l)$ because we had already reached the least fixed point $c0l$ anyway. The goal we have to prove is the following. (We have renamed $\exists c2l, e2l, c2r, e2r$ on the right hand side to $\exists c4l, e4l, c4r, e4r$ to avoid variable name clashes.)

$\exists c4l, e4l, c4r, e4r.$ `iterate` $envl\ e0l\ c1l$ `==` $(c4l, e4l)\ \wedge$
`iterate` $envr\ e0r\ c1r$ `==` $(c4r, e4r)\ \wedge \ldots$

Using `iterate`'s definition, and the induction hypotheses above, this may be rewritten to

$\exists c4l, e4l, c4r, e4r.$ $(c0l, e1l)$ `==` $(c4l, e4l)\ \wedge$
`if ceq` $c1r\ c2r$ `then` $(c2r, e2r)$ `else` $(c3r, e3r)$ `==` $(c4r, e4r)\ \wedge \ldots$

This may be proved easily by a boolean case analysis on `ceq` $c1r\ c2r$ followed by rewriting.

Case 3: `ceq` $c0l\ c1l$ `== false` and `ceq` $c0r\ c1r$ `== true`. The left hand side iterates, and the right hand side fixes. We cannot derive a contradiction here, as we did for the `THMR` case. The same line of attack is used as in case 2. Instead of the right vertical instantiation followed by the use of induction hypothesis 6, we use a left vertical instantiation and induction hypothesis 5.

Case 4: `ceq` $c0l\ c1l$ `== false` and `ceq` $c0r\ c1r$ `== false`. Both sides have not arrived at their fixed points yet. This is a combination of cases 2 and 3 above; first we use a left vertical instantiation of the `THM` induction hypothesis 1. This is followed by a right vertical instantiation of hypothesis 1.

> The `THMI` hypothesis 5 (both the left hand and right hand side change) can then be instantiated. Two boolean case analyses are required to discharge this subgoal.

This concludes the `LET REC` part of the monotonicity proof.

The remainder of the monotonicity proof proceeds without any further complications.

A Post Mortem of the Proof

The proof described above is complex because the induction is not straightforward, and several different induction hypotheses must be instantiated a number of times. The proof could probably be simplified by changing the definition of `reduce` to iterate directly, without using the auxiliary function `iterate` (see page 126.) This would allow us to dispense with the inductions on $n0l$ and $n0r$, because these would be subsumed by the inductions on nl and nr. Although we can remove `THMI` from the statement of `THM`, this introduces a problem concerning well-typedness. At the moment, only `LET REC` expressions with a bottom approximation are well-typed, but using a fixed point computation without `iterate` will introduce non-bottom approximations. This will complicate the statement of the theorem because we have to introduce a less stringent typing function, and apply it when we are inside a fixed point computation. Moreover, it will be harder to state the 'vertical' invariants on the fixed point induction because they are now mixed up with `reduce`, rather than being localised around `iterate`. It seems likely that, although the new proof will contain less nested inductions it will not be very much simpler, because it follows the same principle as the current proof and the notational difficulties are transferred from `iterate` to `reduce`.

The current proof contains a large number of nested inductions and case analyses. Most often, a natural number induction on size, and a structural induction are really aspects of one induction. This induction is more general than the structural induction, but cannot be expressed directly. Moreover, nested inductions on expressions which have the same shape do not introduce 64 subgoals, but only eight, so that this too makes the proof seem more complicated than it really is. We have a total of 10 nested inductions when dealing with the fourth case of the `THMI` subgoal of the `LET REC`:

- natural number induction on *nl* (1 case)
- natural number induction on *nr* (1 case)
- structural expression induction on *e0l* (8 cases)
- structural expression induction on *e0r* (8 cases)
- natural number induction on *n0l* (1 case)
- natural number induction on *n0r* (1 case)
- boolean case analysis on `ceq` *c0l c1l* (2 cases)
- boolean case analysis on `ceq` *c0r c1r* (2 cases)
- boolean case analysis on `ceq` *c1l c2l* (2 cases)
- boolean case analysis on `ceq` *c1r c2r* (2 cases)

Although it seems we could have 1024 cases, we never have more than 17 subgoals in the actual proof.

A complexity inherent in the proof is that we need to use two squares in figure 4–4. That is, we cannot say that every square is ordered because `ple` *e0l e1l* and `ple` *e0r e1r* do not hold. For this reason we have to supply the intermediate outputs of `reduce` (*c0l*`::`*envl*) *e0l* and `reduce` (*c0r*`::`*envr*) *e0r* and the related ordering relations. This makes the statement of `THMI` so unwieldy. An additional problem is that a large number of properties are proved at the same time: `reduce` preserves shape, type, ordering between *e0l* and *e1l*, as well as the ordering between *e1l* and *e1r*. Most of these properties are interdependent, and even if they weren't it would be very tedious to do essentially the same proof over and over again. We intended to make the proof schematic in an extra invariant `P`, which could later be instantiated with a desired operator or function. In principle it would be possible to derive the minimal properties which `P` would need to obey to be used as an invariant on `reduce` and `iterate`. After some initial difficulties we decided it made more sense to try to prove the theorem in the above form first, and maybe later try to extend it. Finally, as most properties must be preserved by `iterate` to be any use for `reduce` they had to be mentioned at least five times. (1) To hold between the inputs of `reduce` (`P#(`*xil*`,`*xir*`)`), (2) to hold between the outputs of `reduce` (`P#(`*x(i+1)l*`,`*x(i+1)r*`)`), (3), (4) similarly for `iterate`, and (5) to hold between inputs and outputs of `iterate` (*e.g.* `P#(`*xil*`,`*x(i+1)l*`)`.)

4.3.3 Some Corollaries

We will now state some results which follow from the monotonicity of `reduce`. The proofs are generally straightforward. The monotonicity theorem as proved above is very unwieldy. Some corollaries stating the monotonicity of `reduce` and `iterate` separately, and the monotonicity in single arguments have also been given.

The following result states that `reduceSeq` is total, monotone, preserves type and shape. The statement is much like `THMR`, and we will not go into any detail explaining it.

```
⊢ ∀envl,envr,e0l,e0r,inp1,inp2,t1,t2.
    lle inp1 inp2 == true ∧
    length inp1 == length inp2 ∧
    (∀i.  i < length inp1 == true →
              typeOfConst (elem inp1 i) == t1 ∧
              typeOfConst (elem inp2 i) == t1) ∧
    lle envl envr == true ∧
    ple e0l e0r == true ∧
    shapeEq e0l e0r == true ∧
    map typeOfConst envl == map typeOfConst envr ∧
    typeOfExpr (t1::map typeOfConst envl) e0l == (t2,true) ∧
    typeOfExpr (t1::map typeOfConst envr) e0r == (t2,true) →
        ∃out1,e1l,out2,e1r.
         lle out1 out2 == true ∧
         (∀i.  i < length out1 == true →
                   typeOfConst (elem out1 i) == t2 ∧
                   typeOfConst (elem out2 i) == t2) ∧
         length out1 == length inp1 ∧
         length out2 == length inp2 ∧
         reduceSeq envl e0l inp1 == (out1,e1l) ∧
         reduceSeq envr e0r inp2 == (out2,e1r) ∧
         ple e1l e1r == true ∧
         shapeEq e1l e1r == true ∧
         shapeEq e0l e1l == true ∧
         shapeEq e0r e1r == true ∧
         typeOfExpr (t1::map typeOfConst envl) e1l == (t2,true) ∧
         typeOfExpr (t1::map typeOfConst envr) e1r == (t2,true))
```

We can generalise the theorem by not requiring *inp1* and *inp2* to be of the same length, but this means we lose `ple` *e1l e1r* `== true` from the conclusion. We also have to modify

```
(∀i.  i < length inp1 == true → typeOfConst (elem inp1 i) == t1 ∧
                                typeOfConst (elem inp2 i) == t1)
```

to

```
(∀i.  i < length inp1 == true → typeOfConst (elem inp1 i) == t1) ∧
(∀i.  i < length inp2 == true → typeOfConst (elem inp2 i) == t1)
```

and a similar change in the conclusion.

An important meta-result is that `iterate` computes a fixed point. We define `FIXPOINT` and `LEASTFIXPOINT` as follows.

```
val FIXPOINT#(c,env,e) = ∃f.  reduce (c::env) e == (c,f);
val LEASTFIXPOINT#(c,env,e) =
    FIXPOINT#(c,env,e) ∧
    (∀d.  typeOfConst d == typeOfConst c →
          FIXPOINT#(d,env, e) → cle c d == true);
```

We proved that `iterate` computes a fixed point provided its second approximation is more defined than its first approximation. This condition guarantees convergence.

```
⊢ ∀envl,c0l,e0l,c1l,e1l,c2l,e2l.
    typeOfExpr (map typeOfConst (c0l ::  envl)) e0l ==
               (typeOfConst c0l,true) ∧
    reduce (c0l ::  envl) e0l == (c1l,e1l) ∧
    cle c0l c1l == true ∧
    iterate envl e0l c0l == (c2l,e2l) → FIXPOINT#(c2l,envl,e0l)
```

If we start with the bottom approximation, we compute the least fixed point:

```
⊢ ∀envl, c0l, e0l, c1l, e1l.
   typeOfExpr (map typeOfConst (c0l::envl)) e0l ==
            (typeOfConst c0l,true) ∧
   ceq (bottomOfConst c0l) c0l == true ∧
   iterate envl e0l c0l == (c1l,e1l) → LEASTFIXPOINT#(c1l,envl,e0l)
```

We can also prove that if *c1l* lies between the first approximation *c0l* and its fixed point *c2l*, then *c2l* is also *c1l*'s fixed point.

An Alternative Dynamic Semantics

We have encoded an alternative dynamic semantics for expressions. It only differs from `reduce` in the `If` statement through its use of the greatest lower bound rather than the bottom value. It is the greatest lower bound semantics `reduceGlb` of Section 3.3.5.

```
fun reduceGlb l (If (e1, e2, e3, ch)) =
    (fn (c1, f1) =>
    (fn (c2, f2) =>
    (fn (c3, f3) =>
            (case match ch c1 of
                  uu => glb c2 c3 |
                  tt => c2 |
                  ff => c3,
             If (f1, f2, f3, ch)))))
            (reduceGlb l e1) (reduceGlb l e2) (reduceGlb l e3) | ...
and iterateGlb l e c = ...
and againGlb l e d = iterateGlb l e d;
```

It delivers the greatest lower bound of the two branches rather than the bottom value when the output of *e1* is not defined enough to decide between the `THEN` and `ELSE` branches. As the bottom value `bottomOfConst` *c2* is smaller than anything, in particular `glb` *c2 c3*, we can deduce that the result of `reduceGlb` is more defined than that of `reduce` on the same inputs. We proved the following result:

$\vdash\ \forall e0l, envl, envr, t.$
    `lle` $envl$ $envr$ `== true` $\wedge$
    `map typeOfConst` $envl$ `== map typeOfConst` $envr$ $\wedge$
    `typeOfExpr (map typeOfConst` $envl$`)` $e0l$ `== (`t`,true)` $\rightarrow$
        $\exists c1l, e1l, c1r, e1r.$
        `reduce` $envl$ $e0l$ `== (`$c1l, e1l$`)` $\wedge$
        `reduceGlb` $envr$ $e0l$ `== (`$c1r, e1r$`)` $\wedge$
        `cle` $c1l$ $c1r$ `== true` $\wedge$
        `ple` $e1l$ $e1r$ `== true` $\wedge$
        `shapeEq` $e1l$ $e1r$ `== true` $\wedge$
        `shapeEq` $e0l$ $e1l$ `== true` $\wedge$
        `shapeEq` $e0l$ $e1r$ `== true` $\wedge$
        `typeOfConst` $c1l$ `==` t $\wedge$
        `typeOfConst` $c1r$ `==` t $\wedge$
        `typeOfExpr (map typeOfConst` $envl$`)` $e1l$ `== (`t`,true)` $\wedge$
        `typeOfExpr (map typeOfConst` $envr$`)` $e1r$ `== (`t`,true)`

Thus `reduce` is a more pessimistic semantics than `reduceGlb`. A similar theorem has been proved relating `iterate` and `iterateGlb`. We also proved the monotonicity result `THM` of `reduce` for `reduceGlb`.

4.3.4 Future Work

One result which would be useful to prove, would be under which circumstances we do not get undefined values as outputs from a circuit. Three conditions come to mind immediately: (1) no undefined inputs, (2) no explicit undefined constants in the circuit description (other than initial values for `LET REC` statements), (3) no delayless feedbacks. This would be a non-trivial theorem to prove because the proof does not reflect the structure of the circuit, and does not seem to lend itself to a structural induction. However, the predicates corresponding to the second and third conditions must be defined structurally on the circuit. A syntactic condition on the circuit structure which expresses this would be very useful. It would allow a simpler `match` function to be used, which can use two-valued boolean logic, and can be translated into equality and truth value predicates. This would allow us to rewrite symbolic evaluations to more tractable expressions (see, for example, Section 5.1.5.)

It would also be interesting to try to implement some other alternative semantics for picoELLA and try to formally prove their interrelationships.

Another area which we have so far ignored is the existence of normal forms for choosers and expressions. Davies proved some results concerning expression normal forms and equivalence checking in [47]. In Section 5.3 we prove some basic properties about behaviour equivalence of expressions.

4.4 Proof Programming and Large Proofs

In this section we will describe our experience with performing large proofs in the interactive proof assistant LAMBDA. Theorem proving in a proof system, especially an interactive proof assistant, often has the flavour of programming. The recent term 'proof engineer' is a good indication of the type of work carried out in a proof system. In common with conventional programming the early days of proof programming were not interactive (*cf.* Stanford LCF [127].) The Edinburgh LCF introduced the meta-language ML so that proofs could be provided interactively [77]. A great leap was made forward by the provision of tactics and tacticals which allowed backward theorem proving [130]. Rather than coding a proof in terms of basic inference rules the proof structure became clearer through the use of higher-order operations such as tactics and tacticals [40]. Tacticals provided sequencing, choice and repetition operators, corresponding to high-level programming language constructs. Paulson's Isabelle [143,142] pioneered the use of combining rules by unifying the conclusion of one rule with the premise of another. LAMBDA uses the same approach to the construction of proofs as Isabelle. However, theorem proving in LAMBDA still has an 'assembler programming' feel to it. Consider the proof of the reflexivity of `cle`.

```
pushGoal pe "G // H ⊢ ∀x.  cle x x == true";
apprl allR;
apprl constInduct;
(* Premise 1:  Cons *)
applyTac cleTac;
apprl eqReflL';
permg[2];
apprl eqReflL';
applyTac (rewriteTac []);
permg[2];
apprl zeroExistsL;
apprl eqTotalR;
(* Premise 2:  CoTuple *)
applyTac cleTac;
val cleReflT = popGoal();
```

This sequence of commands may be interpreted as a program computing the proof for the theorem. All steps are at a low level, but a basic hierarchy is already present through the use of derived rules such as `eqReflL'` (the reflexivity of `eq`.) The use of `cleTac`, and `rewriteTac` also lifts the conceptual level from a purely rule-based level. Using tacticals we can implement this proof by a single tactic.

```
doRules [allR,constInduct] thenT cleTac
thenR eqReflL' thenG [2] thenR eqReflL' thenT (rewriteTac [])
thenG [2] thenRL [zeroExistsL,eqTotalR]
```

(Section B.2 contains some figures about the proportion of permutations and rules, number of rules per tactic, *etc.*) Note however, that there are a number of explicit numbers in the permutation tactics. These correspond to the positions of the hypotheses in the hypothesis lists. This form of absolute addressing limits the reusability of the tactic. Often, the first statement of a theorem is incorrect. This means that we have to amend the statement and rerun the proof, possibly with changes. Adding a hypothesis, for example, results in all absolute addresses being out by one. While this is easy to fix (at the start of the proof we move the new hypothesis to the end of the hypothesis list, so that it does not interfere with the old proof, and we only move it forward when it is required) the proof should be independent of this sort of operation. By providing some sort of symbolic addressing we can make the proof more readable and more robust. We created two functions `permgTac'` and `permhTac'` which take a string, corresponding to a variable name or initial segment of a hypothesis, as input and generate the appropriate absolute address. The above tactic could be rewritten as follows.[4]

```
doRules [allR,constInduct] thenT cleTac thenG' ["r1'"]
thenR eqReflL' thenG' ["r'"] thenR eqReflL' thenT (rewriteTac [])
thenG' ["r'"] thenRL [zeroExistsL,eqTotalR]
```

Non-existence hypotheses can be referred to by a unique initial segment. If `cle r1' r1' == true` and `cle r' r' == true` are two hypotheses, the first can be looked up as `permhTac'["cle r1"]`. A severe drawback of the current system is that it depends on the way in which hypotheses are printed. This is particularly a problem for variables such as `r1'` and `r'`. When a new variable `r` is introduced previously present `r`s may be renamed [64, Section 2.8].[5] A method to unambigu-

[4]This is not quite true, as in our implementation the position of `r1'` and `r'` is defined statically at the start of the tactic. In the example there are no such variables yet, and the tactic will fail with `Exception- PrintStringNotFound "E r1'" raised`. Nevertheless, even with this limitation the current implementation has made proofs considerably more readable.

[5]LAMBDA uses rule connections [64, Section 2.9] to ensure that names don't get arbitrarily renamed when new names are introduced. This makes proof output much more understandable, but does not help us here. A consequence of using the way in which hypotheses are printed out is that proofs of different premises are not commutative. This is due to the fact that variable

ously identify variables and hypotheses is needed. This helps particularly in large proofs such as the monotonicity of `reduce`, described previously. We can have a large number of variables and hypotheses: at one point 51 existence hypotheses and 83 'conventional' hypotheses were present in the first premise alone! It may be useful to provide support to give explicit names such as 'indhyp', 'isbottom' to individual hypotheses so that they may be retrieved by name, rather than their variable position in the hypothesis list, or the manner in which they are printed. This, of course, represents introducing variable names in our proof programming language, which contain the varying location on which the premise is currently situated. The LEGO proof system supports the explicit naming of hypotheses in exactly this manner [116].

As with conventional (interactive) programming, after a proof command is entered it will often fail, either due to type-checking, or the failure of a rule or tactic. A slightly modified version is then tried until a working version is obtained. We advance a number of proof steps and discover that we made an error some steps ago. We have to undo those steps, modify the tactic, and possibly rerun the proof from that point. In such an interactive mode of proving it becomes hard to keep track of old versions of proofs, old versions of the statement of the theorem (most initial statements will be incorrect), *etc.* [105]. At a very basic level a history mechanism to help with the rerunning of (modified) previous commands is useful. At a higher level, proof scripts should be automatically generated. A proof script containing the verbatim input is better than nothing, but we would prefer to have undone commands automatically deleted from the proof script. Some systems allow proof tree fragments to be saved and re-used in other proofs [154]. Such an encapsulation facility would be useful even at a command-line level. Note that a proof fragment does not necessarily correspond to a lemma because a proof fragment may have side effects to expressions which are not captured in a lemma. A replay facility (which includes interactive actions such as selecting expressions

---

names may be dependent on the order in which the proof is performed.

to be unified in a pop-up window) is useful in this context also. At a basic level cutting and pasting of proof scripts works, but a more advanced mechanism should be available. Saxe *et al.* [158] describe the use of annotations in the LARCH rewrite system. If the new proof does not obey the annotations to the proof it is halted at that point, even though the proof script may still be able to (incorrectly) proceed. The process of rerunning proofs which we believe to be correct, but don't want to check at this moment can benefit from recent work by Boulton [20]. A proof may be lazy which allows its execution to be deferred. It can be evaluated off-line afterwards to ensure that the proof is correct, and a proper (non-lazy) theorem is then obtained.

A number of proof systems have provided graphical user interfaces to a proof system. These include the Interactive Proof Environment IPE of Ritchie [154, 153], the VERITAS proof system [87], and the Mural proof environment [134]. A graphical interface *per se* does not resolve our problems, although abilities such as showing the 'shape' of a proof in the form of a (condensed) proof tree are very useful. In our opinion it is crucial that not all interaction with the proof engine needs to be performed via a 'point and click' interface. It must be possible to enter tactics such as those displayed above in one step otherwise the system will become too frustrating to use for experienced users. This corresponds to the ability to enter input in the largest chunks still comprehensible to the user, and at a sufficiently high level of abstraction.

Although version control of theorem statements and their possible proofs have already been mentioned in our opinion this is very important area of research. Proofs for hardware verification are generally large and boring ([41], Section 4.3.2) rather than small and interesting. These sort of problems have been studied for software engineering and they will have to be addressed if theorem provers are to be used on industrial-sized problems. Thus proof support needs to be provided in the form of an integrated proof support environment where proof documentation and annotation, proof dependency graphs, version control, library, and data base support are combined. LAMBDA supports a data base through the use of hierarchical persistent data bases. There is no structuring of data (*e.g.* lemmas, rules, tactics) in this data base other than explicit structure provided by the user. A notion of a theory would be very useful, especially if they may be hierarchical and parametrised. VERITAS [85,87] and HOL [79] both support hierarchical theories. Windley has done some work on providing parametrised theories for HOL [183,184]. OBJ [68] and LEGO [116] both support parametrised theories. Mural allows hierarchical theories, alternative proofs, partial proofs, *etc.* [11,134].

4.5 Conclusions and Future Work

The embedding of picoELLA in LAMBDA turned out to be a sizable effort. This was partly due to the steep learning curve associated with the use of a proof system. Moreover, the monotonicity proof of the dynamic semantics had to be encoded in a non-trivial manner. A number of interesting properties have been proved about the standard picoELLA semantics `reduce`, and one alternative semantics `reduceGlb`. It would be challenging to formalise some of the alternative semantics described in Section 3.3.5 and try to prove some relationships between them.

Unfortunately, due to the different logic of later versions of LAMBDA, the current system is dated. Moreover, if a production quality system is required a larger subset of ELLA must be tackled. It would be better to treat the current system as a prototype, and discard it, rather than try to build on top of it. This cannot be seen as a shortcoming of the current embedding the explicit purpose of which was to explore a minimalist approach to combining a subset of ELLA and a proof system.

Chapter 5

Case Studies

In this chapter we discuss how the embedded semantics for picoELLA may be used in practice. The first section describes how familiar looking operational semantics rules are derived and used as a powerful simulator. In Section 5.2 we show how circuits may be synthesised interactively. We also define a simple hardware synthesis function. Section 5.3 briefly shows how behaviour preserving transformations can be proved correct using the embedding. Finally future work is discussed.

5.1 Operational Semantics Based Symbolic Simulation

In this section we describe how the embedded static and dynamic semantics may be used as one would use a 'paper semantics'. We can check semantic derivations, or use it as a simulator. The work in this section has been presented previously in [73].

Using the definitions of the static and dynamic semantics and the derived properties presented in the previous chapter, the operational semantics rules described in Section 3.3 may be derived within LAMBDA. This is in contrast with recent work which uses the HOL proof assistant to embed a semantics [175]. There the operational semantics rules are the basis from which rewrite rules are derived.

Consider rule reduceSeqCons (3.24 of Section 3.3.3):

$$\frac{c,\ \Gamma \vdash\ program \Rightarrow\ c',\ program' \qquad t,\ \Gamma \vdash\ program' \Rightarrow\ t',\ program''}{c :: t,\ \Gamma \vdash\ program \Rightarrow\ c' :: t',\ program''}$$

It takes the first element of the input stream, and evaluates the circuit with this input. The program is then evaluated with the remainder of the input stream. Recall from page 126 that the function `reduceSeq` implements the time semantics of picoELLA. The relevant clause of the definition of `reduceSeq` is:

```
fun reduceSeq l e (h::t) =
               (fn (c1, f1) =>
               (fn (c2, f2) => (c1::c2, f2))
               (reduceSeq l f1 t)) (reduce (h::l) e) | ...
```

The corresponding rewrite rule which is returned by LAMBDA is

```
⊢ reduceSeq l e (h::t) ===
when#(E t, (fn (c,f) =>
           (fn (d,g) => (c::d,g))
           (reduceSeq l f t)) (reduce (h ::  l) e))
```

(Strictly speaking this is the optimised rule delivered by `processFun` [64, Section 3.2].) From this we would like to derive something resembling the 'paper' version. This is not difficult; but we have to decide on the underlying representation for our embedded semantics rules. The rule `reduceSeqCons` below corresponds to reduceSeqCons above. Names of rules in the `typewriter` font denote the embedded rules, those in roman font the 'paper' rules.

The format of the embedded operational semantics rule below is only one possible convention we could have used. The differences are mainly pragmatic and do not affect any of the results presented here.

```
E instream_, E env_, E circ1_, E t_
⊢ reduceSeq env_ circ1_ instream_ == (outstream_,circ2_)
E (i1_ :: env_), E circ_, E t_
⊢ typeOfExpr (map typeOfConst (i1_ :: env_)) circ_ == (t_,true) ∧
reduce (i1_ :: env_) circ_ == (o1_,circ1_)
--------------------
E (i1_ :: instream_), E env_, E circ_, E t_
⊢ reduceSeq env_ circ_ (i1_::instream_) == (o1_::outstream_,circ2_)
```

To evaluate a program *circ_* with a non-empty input stream *i1_::instream_*, the head of the input stream is pushed onto the environment *env_*. The environment $\Gamma\{(name, c)\}$ in the paper rules is encoded as *(i1_::env_)*. (Strictly speaking `reduceSeqCons` also implements rule 3.26 dealing with the `INPUT` statement.) The expression *circ_* is then evaluated within this time step. The remainder of the input stream is then evaluated using the new circuit *circ1_*. Finally, the output *o1_* is prepended to the resulting output stream.

Note that the first premise deals with both the static and dynamic semantics within one time step. The dynamic semantics operates only on well-typed expressions. As both semantics are defined on the structure of expressions the static and dynamic semantics may be conveniently evaluated simultaneously.

In the remainder of this chapter we will display embedded operational semantics rules in the following format.

```
⊢ (instream_, env_ ⊢ circ1_ ⇒ (outstream_,circ2_))
⊢ (i1_ :: env_ ⊢ circ_ ⇒ (o1_,circ1_) : t_)
--------------------
⊢ (i1_ :: instream_, env_ ⊢ circ_ ⇒ (o1_ :: outstream_,circ2_))
```

This is `reduceSeqCons`. The input streams (*i1_ :: instream_* and *instream_*) are shown for `reduceSeqCons` and `reduceSeqNil` only. A pretty-printer has been implemented which outputs rules in this format. However, as discussed in Sections 4.2.1 and 4.2.2 there are some differences between picoELLA on paper (Section 3.3) and the embedded picoELLA. The de Bruijn encoding of variables is the main change. To increase legibility, expressions have been manually converted from a prefix format (*e.g.* `Let (`*e*`,Index1 (Var 0))`) to infix format (`LET` *e* `IN (Var 0)[1]`.) Constant to expression conversions and (constant) tuple constructors have been omitted where this does not introduce ambiguities. For example, `Delay (CoTuple (`*c*`,`*d*`), Tuple (Var 0,Var 1))` is shown as `DELAY ((`*c*`,`*d*`), (Var 0,Var 1))`. However, as LAMBDA provides no quotation/-anti-quotation facilities, all input must still use the raw syntax.

Certain rules contain premises dealing solely with the static semantics, or only with the dynamic semantics. Consider the rule `reduceIf'` for the multiplexor.

```
[6] ⊢ E t_
[5] ⊢ o3_ == (case match chooser_ out_ of
uu => bottomOfConst o1_ | tt => o1_ | ff => o2_)
[4] ⊢ chooser_ :  t_
⊢ (env_ ⊢ branch2_ ⇒ (o2_,branch2'_) :  t1_)
⊢ (env_ ⊢ branch1_ ⇒ (o1_,branch1'_) :  t1_)
⊢ (env_ ⊢ circ_ ⇒ (out_,circ'_) :  t_)
--------------------
⊢ (env_ ⊢ IF circ_ MATCHES chooser_ THEN branch1_ ELSE branch2_ ⇒
(o3_,IF circ'_ MATCHES chooser_ THEN branch1'_ ELSE branch2'_) :  t1_)
```

Premises `[4]` and `[6]` deal only with the static semantics: the choosers must be well-typed and have (denoting) type *t_*. Note that *branch1_* and *branch2_* must have the same type *t1_*, which is also the type of whole `IF`. In premise 5 the output of the `IF` is computed; this concerns the dynamic semantics only. The three cases (match, no match, and don't know) are represented in the `case` statement by `tt`, `ff`, and `uu` respectively (see Section 4.2.3.) The `IF` is strict; both branches must always be evaluated. Four other rules dealing with the `IF` are particular instantiations of this rule, as we shall see later.

All derived operational semantics rules are listed with a brief explanation in appendix C.

It is important to realise that *circ_*, *t_*, *etc.* are meta-variables. The `reduceIf'` rule is really a rule schema, which may be instantiated in an infinite number of different ways. When it is applied to a particular `IF` statement such as `IF hi MATCHES hi THEN lo ELSE hi`, *circ_*, *chooser_*, *branch1_* and *branch2_* will be unified with `hi`, `hi`, `lo`, and `hi` respectively. This unification is reflected in every place where these variables occur in the rule. The unification works both ways: meta-variables in a rule are unified with the current goal so that the rule applies, but meta-variables in the goal may also be unified (specialised, made more concrete) for the rule to apply. Recall that in LAMBDA meta-variables may be *flexible* or *rigid* (Section 4.1.1.) The former are used to stand for some term to be determined as the proof proceeds, the latter require proofs to be schematic in the variable. Rigid variables ensure that a general result, rather than an instantiation of the result, is proved. Varying the flexibility of variables allows the use of embedded operational semantics rules in various ways, as we shall see below.

5.1.1 A Simple AND Gate

An AND gate may be described in picoELLA as

```
IF e MATCHES (hi,hi) THEN hi ELSE lo
```

or, using the syntax of the embedding:

```
If (e, Const hi, Const lo, T (C hi, C hi));
```

Here e is the input to the circuit. The abbreviations `bit`, `hi`, and `lo` have been defined as `Cons(0,1)`, `Cons(1,1)`, and `Cons(2,1)` respectively. All of `bit`, `hi` and `lo` have type `Type 1`. We will simulate an AND gate with `(hi,lo)` as input using the `reduceIfFf` and `reduceConst` rules. The rule `reduceIfFf` is comparable to the rule `reduceIf'` of the previous page, but always chooses the `ELSE` branch.

```
***** Level 1 *****
⊢ (env_ ⊢ IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo ⇒
 (lo,IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo) :  Type 1)
--------------------
⊢ (env_ ⊢ IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo
⇒ (lo,IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo) :  Type 1)
> apprl reduceIfFf;

***** Level 2 *****
[6] ⊢ E t_
[5] ⊢ match (hi, hi) out_ == ff
[4] ⊢ (hi, hi) :  t_
[3] ⊢ (env_ ⊢ lo ⇒ (lo, lo) :  Type 1)
[2] ⊢ (env_ ⊢ hi ⇒ (o1_, hi) :  Type 1)
[1] ⊢ (env_ ⊢ (hi, lo) ⇒ (out_, (hi, lo)) :  t_)
--------------------
⊢ (env_ ⊢ IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo
⇒ (lo,IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo) :  Type 1)
```

We now have six subgoals to prove, the first of which deals with the input to the `IF`. The second and third subgoals compute the `THEN` and `ELSE` branches respectively. As stated earlier, both branches must be evaluated, because the result circuit is always used to describe the `IF` at the next time step. Note, however, that the value output *o1_* does not appear in the conclusion of `reduceIfFf`. It is for this reason that it has not been constrained to (unified with) a concrete term such as `lo`. The variable *branch1_*, in contrast, did appear in the conclusion of the `reduceIfFf` rule and has been unified with the corresponding expression (`hi`) in the goal it was applied to. The fourth premise states that the chooser must be well-typed; in this case it has type *t_*. Like *o1_*, *t_* is an as yet uninstantiated meta-variable. Evaluating premise 1 forces *t_* to become a tuple type. We also have to prove that the type denotes in premise 6. `[5]` expresses the constraint that we choose the `ELSE` part of the `IF`; the result *out_* of the input circuit must not match with the chooser. We may now apply `reduceTuple` to reduce the tuple in premise 1 to two subgoals. Following this we apply `reduceConst` to premises one to four. (Recall that constants are converted to expressions using `Const`, page 115.)

```
> applyTacn [1,2,3,4] (doRule reduceConst);


***** Level 4 *****
⊢ E (TyTuple (t1_,t2_))
[6] ⊢ match (hi, hi) (hi,lo) == ff
⊢ (hi, hi) :  TyTuple (t1_,t2_)
[4] ⊢ lo :  Type 1
[3] ⊢ hi :  Type 1
[2] ⊢ lo :  t2_
[1] ⊢ hi :  t1_
--------------------
⊢ (env_ ⊢ IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo
⇒ (lo,IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo) :  Type 1)
```

The tactical `applyTacn l t` applies tactic `t` to all premises in the list `l`. The function `doRule` converts a rule into the tactic which applies the rule if it is applicable, and fails otherwise.[1] A similar function `tryRule`, is the identity tactic if the rule fails to apply. Functions `tryRules` and `doRules` operate on rule lists.

As mentioned earlier, the type of the chooser has been constrained to a less general type `TyTuple` (*t1_*,*t2_*). Evaluating premises one and two will specialise it further to `TyTuple (Type 1,Type 1)`, as the type of `hi` and `lo` is `Type 1`. All the subgoals, except `[6]`, are now dealing with the static semantics, or typing of terms.

[1] Unification is used when the rule is applied, not matching. Thus `doRule` is equal to `unifyTac` rather than `matchTac`.

The tactic application `applyTacn [1,2,3,4] (doRule reduceHi elseR reduceLo)` discharges premises one to four. Using `reduceMatchTac`, a tactic which rewrites expressions involving `match`, we prove premise six (of level 4.)

```
> applyTacn [2] reduceMatchTac;


***** Level 6 *****
⊢ E (TyTuple (Type 1,Type 1))
⊢ (hi, hi) :  TyTuple (Type 1,Type 1)
--------------------
⊢ (env_ ⊢ IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo
⇒ (lo,IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo) :  Type 1)
```

The tactic `doRules [r1,r2]` applies rule `r1` and then applies `r2` to all resulting subgoals. Thus `reduceC` and `reduceHi` are applied to both subgoals resulting from `reduceT`.

```
> applyTac (doRules[reduceT,reduceC,reduceHi]);


***** Level 7 *****
⊢ E (TyTuple (Type 1,Type 1))
--------------------
⊢ (env_ ⊢ IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo
⇒ (lo,IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo) :  Type 1)
> applyTac (doRules[reduceTyTuple,reduceType,reduceSn,reduce0]);


***** Level 8 *****
--------------------
⊢ (env_ ⊢ IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo
⇒ (lo, IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo) :  Type 1)
> val example1a = popGoal();
val example1a = ?  :  rule
```

The theorem is saved as `example1a` so that we can apply this derivation in one step in the future.

While this is very instructive, it becomes tedious very quickly to do this sort of proof by hand. Tactics may be used to great advantage in this sort of regular reasoning. The whole previous example could have been done using one general purpose tactic:

```
val OpSemTac = (repeatT (nonTrivT (tryRules OpSemRules))) thenT
               (tryT (theoremT reduceMatchTac)) thenT
               (tryT (theoremT reduceTypeTac));
applyTac OpSemTac;
```

This tactic repeatedly applies one or more of the standard operational semantics rules until none apply. It then applies `reduceMatchTac` followed by `reduceTypeTac`, to rewrite any typing subgoals. These last two tactics are applied to a subgoal only if they discharge it.

The circuit as it stands is not very useful, as it deals with only one particular input. Moreover, we had to supply the output from the simulation at the start! We will now quickly redo the example, but using meta-variables as output. These will be flexible, so that they may be instantiated as we compute the output for a particular answer. We will also use an abbreviation for the AND gate. Recall that abbreviations are meta-level functions in the proof system (Section 4.1.1.)

```
val bit = Cons (0,1);
val hi = Cons (1,1);
val lo = Cons (2,1);
val AND#(e) = IF e MATCHES (hi,hi) THEN hi ELSE lo;
```

When a new goal is to be proved, all meta-variables are rigid; they cannot be (inadvertently) instantiated. In general this is what is required, because the result so proved is then more general. Every operational semantics rule has meta-variables such as $env\_$, $circ\_$, and $t\_$, which are unified with the corresponding expressions in the premise it is applied to. In this case we want to specialise the meta-variables *out_*, *newcirc_*, and *t_* if required, so we make them flexible using the `flex` command. A pop-up menu shows the current subgoal, and one selects subterms by clicking on them with a mouse. The `flex` command is then automatically generated by LAMBDA, so that it may be included in proof scripts for later use. First we unfold the abbreviation for AND.

```
> (* flex *)
***** Level 2 *****
⊢ (env_ ⊢ AND#(circ_) ⇒ (out_, newcirc_:  t_)
--------------------
⊢ (env_ ⊢ AND#(circ_) ⇒ (out_,newcirc_) :  t_)
> apprl ANDU;

***** Level 3 *****
⊢ (env_ ⊢ IF circ_ MATCHES (hi,hi) THEN hi ELSE lo ⇒
(out_, IF circ'_ MATCHES (hi,hi) THEN hi ELSE lo :  t_)
--------------------
⊢ (env_ ⊢ AND#(circ_) ⇒ (out_,AND#(circ'_)) :  t_)
```

Note that the unfolding of the AND abbreviation affects the meta-variable *newcirc_*. It is instantiated with `AND#(`*circ'_*`)` because it is flexible. This is exactly what we want in this case; we do not want abbreviations to become expanded from one time step to the next. One must be careful in using unification when flexible meta-variables are present. Any rule which unifies with a particular meta-variable will instantiate it; often unnecessarily or incorrectly. Tactics such as `safeOpSemAllTac'` on page 185 take great care to avoid this. A third rule for the `IF` statement `reduceIf` can now be applied.

```
> apprl reduceIf;

***** Level 4 *****
⊢ E t_1
⊢ (hi,hi) :  t_1
⊢ (env_ ⊢ lo ⇒ (o2_,lo) :  t_)
⊢ (env_ ⊢ hi ⇒ (o1_,hi) :  t_)
⊢ (env_ ⊢ circ_ ⇒ (out_,circ'_) :  t_1)
--------------------
⊢ (env_ ⊢ AND#(circ_) ⇒ (case match (hi,hi) out_ of
uu => bottomOfConst o1_ | tt => o1_ | ff => o2_,AND#(circ'_)) :  t_)
```

The rule `reduceIf` defers the computation of the output of the `IF` by delivering a *symbolic answer*. In this case, however, we would like to have a concrete value answer rather than an expression describing what happens in the most general case. After undoing everything using `undoAll()` we prove the result we want by flexing type $t\_$ and circuit $newcirc\_$, expanding all the abbreviations using `applyTac (doRules[ANDU,hiU,loU,bitU])`, finally followed by `applyTac OpSemTac`. The tactic `OpSemTac` uses the rule `reduceIf'` rather than `reduceIf`, so that the required answer is obtained.

```
***** Level 5 *****
⊢ out_ == (case match (Cons (1,1), Cons (1,1)) out_1 of
uu => bottomOfConst (Cons (1,1)) |
tt => Cons (1,1) | ff => Cons (2,1))
⊢ (env_ ⊢ circ_ ⇒ (out_1,circ'_) :  TyTuple (Type 1,Type 1))
--------------------
⊢ (env_ ⊢ AND#(circ_) ⇒ (out_,AND#(circ'_)) :  Type 1)
> val reduceAND = popGoal();
val reduceAND = ? :  rule
```

The abbreviations for `hi` *etc.* have been expanded so that this derived rule `reduceAND` may be used in general contexts without any extra work. This derived rule may be thought of as abbreviating the whole proof tree which was generated to prove this rule. Derived rules may be used very effectively in a hierarchical manner. Simulations may be speeded up by passing rules which reduce subcircuits such as AND gates or adders in one step. An alternative approach, is to write a tactic with the same effect. A tactic would actually *replay* or recreate the proof tree, which would be as slow as rerunning the proof. The application of a derived rule, in contrast, is as fast as the application of a primitive rule.

5.1.2 Adding Time

All of the computations we have shown so far have been within a single clock tick or time step. The following example shows how a delayed AND gate may be simulated during two time steps. At every time step the value at the head of the input stream is put on top of the stack. The expression `Var 0` indicates the first value on the stack or environment *env_*. `DELAY` (c, e) is a unit delay of expression e. Thus the circuit `DELAY (bit,AND#(Var 0))` is an AND gate which takes its input from the input stream, and whose output is delayed by one time step. (`reduceSeqCons` was shown on page 173.)

```
> apprl reduceSeqCons;


***** Level 3 *****
⊢ ([(bit,lo)], env_ ⊢ circ1_ ⇒ (outstream_,newcirc_))
⊢ ((lo,lo) :: env_ ⊢ DELAY (bit,AND#(Var 0)) ⇒ (o1_,circ1_): t_)
--------------------
⊢ ([(lo,lo), (bit,lo)], env_ ⊢
DELAY (bit,AND#(Var 0)) ⇒ (o1_ :: outstream_,newcirc_))
```

```
> apprl reduceDelay;


***** Level 4 *****
⊢ ([(bit,lo)], env_ ⊢
DELAY (out_,circ'_) ⇒ (outstream_,newcirc_))
[2] ⊢ bit :  t_
[1] ⊢ ((lo,lo) ::  env_ ⊢ AND#(Var 0) ⇒ (out_,circ'_) :  t_)
--------------------
⊢ ([(lo,lo), (bit,lo)], env_ ⊢
DELAY (bit,AND#(Var0)) ⇒ (bit ::  outstream_,newcirc_))
```

Note that the output from the circuit is known even though the output from the AND has not been computed yet. We reduce premise 1 using `reduceAND` so that *out_* becomes `lo`, and reduce the second premise using `reduceBit`.

```
> apprl reduceBit;


***** Level 7 *****
⊢ ([(bit,lo)], env_ ⊢
DELAY (lo,AND#(Var 0)) ⇒ (outstream_,newcirc_))
--------------------
⊢ ([(lo,lo), (bit,lo)], env_ ⊢
DELAY (bit,AND#(Var 0)) ⇒ (bit ::  outstream_,newcirc_))
```

We have now completed time zero, and can compute the next time step. Note that the description of the delay now has state `lo`, which was the output from the AND gate at the previous time step. The second time step may be dealt with in exactly the same manner, resulting in the following:

```
***** Level 10 *****
⊢ ([], env_ ⊢ DELAY (lo,AND#(Var 0)) ⇒ (outstream_,newcirc_))
--------------------
⊢ ([(lo,lo), (bit,lo)], env_ ⊢
DELAY (bit,AND#(Var 0)) ⇒ (bit :: lo :: outstream_,newcirc_))
```

The final application of `reduceSeqNil` closes the input stream. Note that only at this point do we know what the final circuit looks like, in case we want to continue this simulation.

```
> apprl reduceSeqNil;


***** Level 11 *****
--------------------
⊢ ([(lo,lo), (bit,lo)], env_ ⊢
DELAY (bit,AND#(Var 0)) ⇒ ([bit,lo],DELAY (lo,AND#(Var 0))))
```

As in the previous example, we could have done all of this with the application of a single tactic `safeOpSemAllTac' [reduceAND]`. This is a quite involved tactic which contains an outer loop for each time step of the simulation. This loop finishes when no more changes have been made to any of the subgoals. Firstly, rules involving time are tried, followed by repeated applications of the standard operational semantics rules excluding those involving `CoTuple`, `Cons`, `TyTuple`, and `Type`. When no more rules are applicable, `reduceCoTupleTac` is used. It applies `reduceCoTuple`, but only if no flexible meta-variables will instantiated as a result of this. If the type is already constrained to a `TyTuple` then `reduceCoTuple` can also be applied safely. Similar tactics for `Cons`, `TyTuple`, and `Type` are then tried. `reduceTypeTac`, which uses the general rewrite system, is applied only if it discharges the premise. Finally, `reduceSafeEqRhsTac` rewrites subgoals of the form `expr == case match ...` in a safe way. The tactical `cutThenT` only retains the first unification of its first argument; this decreases the amount of memory which is used, as well as the execution time. Using two nested loops, rather than a single loop forces the evaluation to take place a single clock tick at a time. This dramatically increases the tactic's speed due to the fact that many small expressions are handled more effectively than one large one. `safeOpSemAllTac'` assumes that the subgoals it operates on are in the operational semantics format, described earlier in this section. The tactics `reduceTypeTac` and `reduceSafeEqRhsTac` are only applied if they discharge the subgoal. If this was not the case, the subgoal could be left in an incorrect format, which would slow down the rewriting. Using a fixed format means that only a limited number of customised rewrite rules are used, rather than the general rewrite tactics provided in the LAMBDA library.

```
fun safeOpSemAllTac' l =
      repeatCutT (nonTrivT (
            (tryRules [reduceSeqNil,reduceSeqCons]) cutThenT
            (tryT (repeatCutT (nonTrivT (
                  (tryRules (l @ safeOpSemRules)) cutThenT
                  (tryTacs [reduceCoTupleTac, reduceConsTac,
                            reduceETyTupleTac, reduceETypeTac,
                            (theoremT reduceTypeTac),
                            (theoremT reduceSafeEqRhsTac)])))))));
val safeOpSemAllTac = safeOpSemAllTac' [];
```

Note that a list of derived rules l may be passed into the tactic. This means that an AND gate is reduced using one derived rule application, rather than a series of primitive rules. This facilitates faster, hierarchical simulation because a circuit does not need to be flattened out into individual gates to be simulated. A smaller memory usage is one of the practical advantages. It is also easier to pinpoint errors in a circuit when it is simulated hierarchically because boundaries of subcircuits are clearer when the subcomponents have not been flattened out. One needs to open up a subcircuit only when it is found to be in error.

5.1.3 Two Parity Checkers

Boulton *et al.* illustrate their approach to the verification of ELLA designs with a parity checker [19]. It consists of two multiplexors, two delays and a NOT gate.

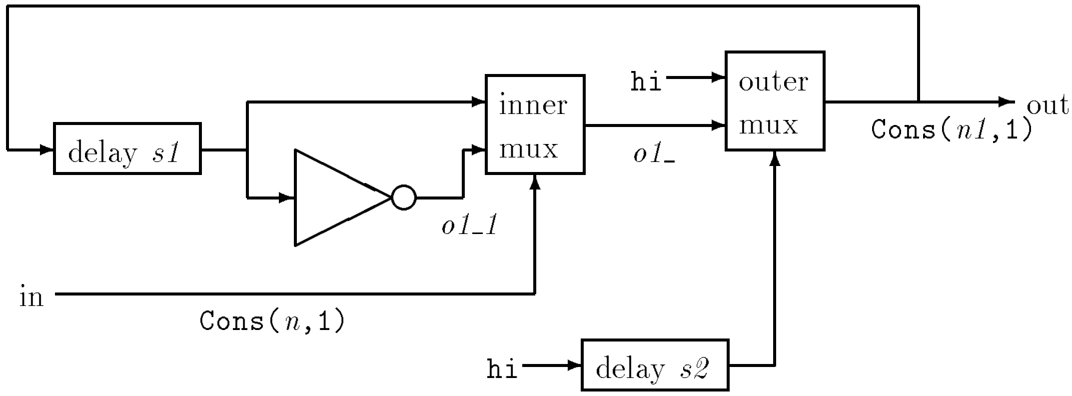

Figure 5–1: The `HOLPC` Parity Checker.

`HOLPC` describes the same circuit as `PARITY_IMP` in the cited paper. The annotations `Cons(`*n*`,1)`, `Cons(`*n1*`,1)`, *o1_*, and *o1_1* correspond to terms in the `reduceHOLPC` rule, discussed below. *s1* and *s2* represent the state of the delays.

```
val NOT_g#(e)        = IF e MATCHES hi THEN lo ELSE hi;
val MUX#(e,b1,b2)    = IF e MATCHES hi THEN b1 ELSE b2;
val REG#(c,e)        = DELAY (c,e);
val HOLPC#(s1,s2,e) =
         LET e IN
         LET INIT bit REC
         (* Use a LET to avoid duplicating register *)
         LET REG# (s1,Var 0) IN
             MUX# (REG# (s2, hi),
                   MUX# (Var 2, NOT_g#(Var 0), Var 0),
                   hi) IN
             Var 0;
```

We use `NOT_g` because `NOT` is the truth value negation operator in LAMBDA. It is worth noting that the state (*s1*,*s2*) of the parity checker is explicit in the abbreviation. The abbreviation may therefore be used in all possible states, and not just the initial state. Most of the complexity of this circuit is due to the restriction that delays must have `lo` as their initial state. We will return to this point in Section 5.3.

```
[5] ⊢ ceq bit (Cons (n1,1)) == false
(* Outer MUX *)
⊢ Cons (n1,1) == (case match (Cons (1,1)) (Cons (b,1)) of
uu => bottomOfConst o1_ | tt => o1_ | ff => hi)
(* Inner MUX *)
⊢ o1_ == (case match (Cons (1,1)) (Cons (n,1)) of
uu => bottomOfConst o1_1 | tt => o1_1 | ff => Cons (a,1))
(* NOT *)
⊢ o1_1 == (case match (Cons (1,1)) (Cons (a,1)) of
uu => bottomOfConst lo | tt => lo | ff => hi)
⊢ (env_ ⊢ circ_ ⇒ (Cons (n,1),h) :  Type 1)
--------------------
⊢ (env_ ⊢ HOLPC#(Cons (a,1),Cons (b,1),circ_) ⇒
(Cons (n1,1),HOLPC#(Cons (n1,1),hi,h)) :  Type 1)
```

The derived rule `reduceHOLPC` contains some points of interest. First note that only the two multiplexors and the NOT gate are present as subgoals; both delays have disappeared. As described in [19], the rôle of the innermost register is to output `lo` at time zero, and `hi` ever after. This is evident from the conclusion of the rule, where the state *s2* is always `hi` after an evaluation. Also note that the output `Cons(`*n1*`,1)` is duplicated in the first register, so that it can be used in the next time step, using the feedback.

At time zero, the values in the registers are both `lo`. In fact, the value in the first delay at time zero is irrelevant:

```
> applyTacAll (doRule reduceDummyVar thenT typeOfChooserTac thenT
                typeOfConstTac thenT reduceTypeTac);

***** Level 5 *****
--------------------
⊢ ([Cons (y,1)], env_ ⊢ HOLPC#(Cons (x,1),lo,Var 0) ⇒
([hi],HOLPC#(hi,hi,Var 0)))
```

The rule `reduceDummyVar` removes subgoals which compute the value of variables which do not contribute to the output of the circuit. This derivation uses an arbitrary input `Cons(`y`,1)` and state `Cons(`x`,1)` in the first delay. The only constraint on these *don't care* values is that they must have the right type. Note that their possible value includes the undefined or *don't know* value. This simulation shows that the state of the new circuit is fully defined no matter what the input at time zero is. In other words, the value of the input at time zero is ignored. This parity checker outputs `hi` at time t if there have been an even number of `hi`s in the input stream from time *one* to time t inclusive.

An alternative parity checker is listed below. It consists of a XOR gate which has its delayed output fed back into itself.

```
val XOR#(e) = IF e MATCHES (hi,lo)|(lo,hi) THEN hi ELSE lo;
val PC#(s,e)= LET e IN
              LET INIT bit REC
              REG# (s, XOR# ((Var 0, Var 1))) IN
                   Var 0;
```

The initial state must be `hi`. `PC` outputs `hi` at time $t+1$ if there have been an even number of `hi` values in the input stream from time *zero* to time t. The output at time zero is `hi`. Given below is an example derivation of `PC`.

```
> applyTac (safeOpSemAllTac'[reducePC]);


***** Level 4 *****
--------------------
⊢ ([hi,lo,hi,hi,lo,lo], env_ ⊢ PC#(hi,Var 0) ⇒
([hi,lo,lo,hi,lo,lo],PC#(lo,Var 0)))
```

Apart from the output at time zero, this output is the complement of that of `HOLPC`, which ignores the initial `hi` at time zero:

```
> applyTac (safeOpSemAllTac'[reduceHOLPC]);


***** Level 4 *****
--------------------
⊢ ([hi,lo,hi,hi,lo,lo], env_ ⊢ HOLPC#(lo,lo,Var 0) ⇒
([hi,hi,lo,hi,hi,hi],HOLPC#(hi,hi,Var 0)))
```

Using conventional verification techniques we proved that the `PC` circuit does indeed count the number of `hi` values in the input stream.

```
(* Number of vs in the input stream from time 0 up to time t.  *)
fun noof v input 0 = 0 |
    noof v input (S t) = if input t = v then (noof v input t) + 1
                                         else (noof v input t);
fun even n = n mod 2 = 0;
fun absinv true = hi | absinv false = lo;
fun state input t = absinv (even (noof hi input t));
```

The function `noof` counts the number of v values in the input stream, `even` returns `true` if there have been an even number of them, and `absinv` is the inverse data abstraction function, mapping booleans to constants. These three functions are combined by `state`, to make the result more readable.

$$
\begin{array}{l}
\vdash \forall l, e, input. \\
\quad (\forall t.\ \ \texttt{reduce}\ l\ (e\ t)\ \texttt{==}\ (input\ t, e\ (\texttt{S}\ t))\ \wedge \\
\qquad\quad (input\ t\ \texttt{==}\ \texttt{hi}\ \vee\ input\ t\ \texttt{==}\ \texttt{lo}))\ \rightarrow \\
\qquad\qquad\qquad \forall t.\ \ \texttt{reduce}\ l\ (\texttt{PC\#(state}\ input\ t, e\ t))\ \texttt{==} \\
\qquad\qquad\qquad\qquad (\texttt{state}\ input\ t,\ \texttt{PC\#(state}\ input\ (\texttt{S}\ t), e\ (\texttt{S}\ t)))
\end{array}
$$

In other words, assuming that a circuit e defines a signal *input* which is `hi` or `lo` at every time step, the behaviour of the parity checker operating on *input* is that of `state` *input*. Thus the first value of the output tuple is `hi` if there have been an even number of `hi` values in the input stream. The second part states that the state of the new circuit is given by the `state` function at time $t+1$. We will have more to say about verifying circuits in this more conventional manner in Section 5.5.

5.1.4 Feedback Loops

In Section 4.3.3 we proved that the operational semantics for picoELLA computes the least fixed point solution of the circuit. An iterative method is used, and the number of iterations may vary to reach the fixed point. This number depends on the circuit but may also vary per input. In the case of delayed feedbacks, however, it takes at most one iteration.[2] This follows from the semantics of the `DELAY` construct, which outputs the same value during one time step. Whatever goes into the delay (*i.e.* the feedback) does not matter. If the output of the defining expression in a delayed `LET REC` is undefined the fixed point is the bottom value, which is reached immediately. It follows therefore that if the output is not undefined, exactly one iteration is needed. (This is an outstanding theorem which would be nice to prove formally in LAMBDA.)

So far we have not explicitly addressed this problem in the embedded operational semantics. In the derived rules for `PC` and `HOLPC` it was assumed that no undefined values were input to the circuit. Premise `[5]` of rule `reduceHOLPC` on

[2] See page 193 for clearer statement of what constitutes an iteration.

page 187 states that the initial approximation `bit` is not equal to the next approximation (`Cons (`*n1*`,1))`. In other words, we do not reach a fixed point after one iteration. This is only so if the input to the circuit is defined, *i.e.* does not contain the undefined value `bit`. The theorem on page 190 which states that `PC` implements a parity checker contains this assumption as an explicit hypothesis.

In the case of delayless feedbacks we cannot rely on this technique. This introduces difficulties in the embedded operational semantics. When using the operational semantics in the form of a rewrite system this poses no problems due to the formulation of the rewrite tactics. We will sketch why this is the case. Recall the definition of the iterative fragment of the `reduce` function:

```
fun iterate l e c =
            (fn (d, f) => case ceq c d of
                               true => (d, f) |
                               false => again l e d)
                         (reduce (c::l) e)
and again l e d = iterate l e d;
```

Were we to omit the definition of `again` we would obtain an unusable rewrite rule for `iterate`. The right hand side would contain an occurrence of `iterate`, and the rewrite system would not converge. We therefore introduce a dummy function `again` to shield the recursive call of `iterate`. We use it as follows.

```
fun reduceNAllTac n = repeatnCutT n
                          (nonTrivT (reduceAllTac cutThenT
                                     iterateTac cutThenT
                                     againTac));
```

This tactic computes n iterations by unfolding all calls to `reduce` and its subfunctions. If we have reached a fixed point the subsequent rewriting of `iterate` will have no effect; otherwise we advance to the next iteration. The function `again` ensures that we only advance one iteration and not an infinite number of iterations. We can compute the least fixed point of any circuit without any problems by using this iteration function.

In the embedded operational semantics this does not work due to the format of the standard operational semantics rules. Consider the three rules which implement the fixed point computation. Rules `reduceLetRec`, `reduceFix` and `reduceIterate` correspond to rules 3.27, 3.28 and 3.29 of Section 3.3.3 respectively. These rules mimic the 'paper' operational semantics rules very well. Some extra premises implement the static semantics which is also incorporated in the embedded semantics rule. For example, the third premise states that the initial approximation must be equal to the bottom value. `reduceLetRec` initiates the recursion in `[1]`, and uses the fixed point in the second premise.

```
***** reduceLetRec *****
⊢ E t1_
⊢ o1_:  t1_
⊢ initial_:  t1_
[3] ⊢ bottomOfConst initial_ == initial_
[2] ⊢ (o1_ ::  env_ ⊢ circ2_ ⇒ (o2_,circ2'_):  t2_)
[1] ⊢ (initial_, env_ ⊢ circ1_ ⇒ (o1_,circ1'_):  t1_)
--------------------
⊢ (env_ ⊢ LET INIT initial_ REC circ1_ IN circ2_ ⇒
(o2_,LET INIT initial_ REC circ1_' IN circ2_'):  t2_)
```

Where $(c,\ l \vdash\ e \Rightarrow \ldots)$ is an abbreviation for `iterate` $l\ e\ c$. Rules `reduceFix` and `reduceIterate` implement the fixed point computation. `reduceFix` detects a fixed point *initial_*; it applies only if the current approximation *initial_* is equal to (unifiable with) the next approximation.

```
***** reduceFix *****
⊢ (initial_ ::  env_ ⊢ circ1_ ⇒ (initial_,circ1'_):  t1_)
--------------------
⊢ (initial_, env_ ⊢ circ1_ ⇒ (initial_,circ1'_):  t1_)
```

The rule `reduceIterate` computes a new approximation *o1_* using the current approximation *initial_*. The third premise of `reduceIterate` states that they are not equal, and hence have not reached a fixed point. Premise `[2]` therefore continues the computation, this time with the new approximation *o1_*. Recall that `ceq` is an encoding of equality on constants.

```
***** reduceIterate *****
[3] ⊢ ceq initial_ o1_ == false
[2] ⊢ (o1_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_): t1_)
[1] ⊢ (initial_ :: env_ ⊢ circ1_ ⇒ (o1_,circ1'_): t1_)
--------------------
⊢ (initial_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_): t1_)
```

We have derived `reduceIterate`*i* from these rules, which are *i* `reduceIterate` applications followed by a `reduceFix`.

As an example of the variation in the number of iterations which may be required consider a flip-flop constructed from two cross-coupled NAND gates. The NAND gates are composed of a NOT and an AND gate.

```
val FF#(e) = LET e IN
             LET INIT (bit,bit) REC
                 (NOT_g#(AND#(((Var 1)[1],(Var 0)[2]))),
                  NOT_g#(AND#(((Var 1)[2],(Var 0)[1])))),
             IN
                 Var 0;
```

In the table below we list the input, output and the number of iterations it takes the flip-flop to stabilise. These iterations are the result of the fixed point algorithm, and do not necessarily have any relation to the relative times a real implementation would take to settle down. Zero iterations means that the circuit reaches its fixed point immediately. In other words, the bottom value is the fixed point. The number of iterations is equal to the number of `reduceIterate` applications. Strictly speaking computing n iterations entails evaluating the circuit $n+1$ times because `reduceFix` must check the fixed point.

Input to `FF`	Output of `FF`	Number of Iterations Required
`(bit,bit)`	`(bit,bit)`	0
`(bit,hi)`	`(bit,bit)`	0
`(hi,bit)`	`(bit,bit)`	0
`(hi,hi)`	`(bit,bit)`	0
`(bit,lo)`	`(bit,hi)`	1
`(lo,bit)`	`(hi,bit)`	1
`(lo,lo)`	`(hi,hi)`	1
`(hi,lo)`	`(lo,hi)`	2
`(lo,hi)`	`(hi,lo)`	2

Figure 5–2: A Derived Truth Table for a Flip-Flop.

The operational semantics derivation for this table is very simple if the required number of iterations for each input are known in advance.

```
(* Split the derivation into individual time steps.  *)
applyTac (repeatT (nonTrivT (tryRules
                   [reduceSeqCons,reduceSeqNil])));
(* Inputs 0,1,2,3:  fix *)
applyTacn [1,2,3,4] (safeOpSemAllTac' [reduceFF0]);
(* Inputs 4,5,6:  iterate once *)
applyTacn [1,2,3] (safeOpSemAllTac' [reduceFF1]);
(* Inputs 7,8:  iterate twice *)
applyTacn [1,2] (safeOpSemAllTac' [reduceFF2]);
```

The rule `reduceFF`i evaluates i iterations and then fixes. Of course, if we do not know the number of iterations in advance, we have to proceed an iteration at a time. That is, we apply `reduceIterate` and try `reduceFix`. If we can discharge the premises of `reduceFix` we have a fixed point, otherwise we have to undo the `reduceFix` and apply another `reduceIterate`. The iteration and fix rules have an identical first premise, which means that they cannot be distinguished early in the derivation. We must evaluate the circuit before we can decide whether we have a fixed point or not.

We have therefore proved three different rules which ease this process. The problem with the `reduceFix` and `reduceIterate` rules is that they have a common initial premise. This premise computes the current iteration. Only after it has been derived do we know whether we applied the correct rule. If we didn't we have to backtrack. The solution therefore, is to factor out this common prefix into a separate rule. After computing the new approximation, the previous and new approximations are passed into the next rule as a hypothesis. We can then rely on the built-in unification to decide whether to apply the iterate or fix rule. First we need an auxiliary function `suspend`.

```
fun suspend l circ c (d,e) = if ceq c d then (d,e)
                                         else iterate l circ d;
```

The term `suspend` l $circ$ c (d,e) is pretty printed as $(c,\ d,\ l \vdash circ \Rightarrow \ldots)$. The rule `reducePrefix` carries out the operations common to `reduceFix` and `reduceIterate`. This operation (*i.e.* the computation of the defining expression, in premise one) is saved in the hypothesis list of the second premise. If the computation of the `LET REC` has reached a fixed point *initial_* and *o1_* will be equal. `reduceFix'` takes advantage of this fact by using the same variable for both. The overall effect of this is that `reduceFix'` only unifies with the second premise of `reducePrefix` if a fixed point has been reached. Thus `reducePrefix` followed by `reduceFix'` is equal to the original `reduceFix`. `reduceIterate'` can always be applied and must therefore be tried after `reduceFix'`. As expected, `reducePrefix` followed by `reduceIterate'` is equal to the original `reduceIterate`.

```
***** reducePrefix *****
[2] 1:  (initial_ ::  env_ ⊢ circ1_ ⇒ (o1_,circ1'_) :  t1_)
⊢ (initial_, o1_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_) :  t1_)
[1] ⊢ (initial_ ::  env_ ⊢ circ1_ ⇒ (o1_,circ1'_) :  t1_)
--------------------
⊢ (initial_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_) :  t1_)
```

The computation of the first premise is passed on to the second premise in the hypothesis list. The hypothesis of the `reduceFix'` rule requires that the first and second approximations are equal, or strictly speaking unifiable, in the rule `reduceFix'` is applied to.

```
***** reduceFix' *****
--------------------
1:  (initial_ ::  env_ ⊢ circ1_ ⇒ (initial_,circ1'_) :  t1_)
⊢ (initial_, initial_, env_ ⊢ circ1_ ⇒ (initial_,circ1'_) :  t1_)
```

```
***** reduceIterate' *****
[2] ⊢ ceq initial_ o1_ == false
⊢ (o1_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_) :  t1_)
--------------------
1:  (initial_ ::  env_ ⊢ circ1_ ⇒ (o1_,circ1'_) :  t1_)
⊢ (initial_, o1_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_) :  t1_)
```

Note that although `reduceIterate'` is always applicable, but that `[2]` will be unsatisfiable if we iterate too often.

The following tactic is able to cope with recursive circuit descriptions and iterates the minimum number of times to find the least fixed point of the circuit.

```
fun safeOpSemRecAllTac' l =
    repeatCutT (nonTrivT (
      (tryRules [reduceSeqNil,reduceSeqCons]) cutThenT
      (tryT (repeatCutT (nonTrivT (
               (tryRules [reduceLetRec,reducePrefix]) cutThenT
               (tryT (repeatCutT (nonTrivT (
                        (tryRules (l @ safeOpSemRules)) cutThenT
                        (tryTacs [reduceCoTupleTac,reduceConsTac,
                                  reduceETyTupleTac,reduceETypeTac,
                                  (theoremT reduceTypeTac),
                                  (theoremT reduceCeqTac),
                                  (theoremT reduceBottomTac),
                                  (theoremT reduceSafeEqRhsTac)])
                        )))) cutThenT
               (tryRules [reduceFix',reduceIterate'])))))));

val safeOpSemRecAllTac = safeOpSemRecAllTac' [];
```

5.1.5 Hierarchical Simulation

Hierarchical simulation is supported in two ways. We have already encountered one method: the use of derived operational semantics rules, coupled with the use of abbreviations. Alternatively, the specification of a circuit can be used.

Derived Operational Semantics Rules

One manner in which operational semantics based simulation may be speeded up is the use of derived rules, such as `reduceAND` and `reduceHOLPC`. An abbreviation for a circuit such as `HOLPC` means that it is shielded from reduction using the standard operational semantics rules. It must be reduced using the specialised `reduceHOLPC` rule, which collapses the standard reduction into one rule. Although these rules increase the simulation speed considerably, the number of premises for each rule is equal to the number of basic gates of the component. For example, the `HOLPC` parity checker consists of two multiplexors, two delays, and one NOT gate. Although `reduceHOLPC` rule has eliminated the delays, there is a `case` statement for each of the remaining gates. A `case` statement contains the computation which represents the behaviour of the gate. It would be nice if we had some sort of support for reasoning about matching. However, due to the data ordering on constants, and the presence of a bottom element in particular, this turns out to be hard to implement. We would like to be able to transform a premise such as

```
⊢ o1_1 == (case match hi (Cons (a,1)) of
uu => bottomOfConst lo | tt => lo | ff => hi)
```

into something like this

```
⊢ o1_1 == if (Cons (a,1)) = hi then lo else hi
```

This can only be done if there are no undefined values present. For inputs this can be assured by providing a premise to this effect. For intermediate values, this depends on the circuit. As long as there are no undelayed feedback wires undefined values cannot arise. Assuming that we have premises in a nice format, we would like to combine them into a single functional expression describing the behaviour of the circuit. This extends naturally to the use of the specification of a circuit instead of its implementation.

Using Specifications

The second manner in which hierarchical simulation is supported is more interesting. It entails using the specifications of circuits rather than their implementations. This is a well-known technique from hardware verification, and has to a limited extent also been used in simulators. Behavioural simulation is supported by a number of simulators, but it does not have the same power. The reason for this is that there is no formal correspondence between the specification and its implementation. In our formal framework the implementation and specification must have the same behaviour to be used in such a manner.[3] Moreover, in informal simulators, the specification must be an HDL program, *i.e.* it must be an algorithm. Our specifications can be arbitrary higher-order logic formulae, which may or may not represent an explicit computation. If it doesn't the simulation will probably need some advice on how to proceed when it arrives at the specification, although some work has been done by Camilleri on executing higher-order logic specifications [33].

To illustrate this approach we define a full adder. ADD is composed of two half adders in the following manner:

```
val AND#(e) = IF e MATCHES (hi,hi) THEN hi ELSE lo;
val OR#(e)  = IF e MATCHES (lo,lo) THEN lo ELSE hi;
val XOR#(e) = IF e MATCHES (hi,lo)|(lo,hi) THEN hi ELSE lo;
val HA#(e)  = LET e IN (XOR#(Var 0), AND#(Var 0));
val ADD#(e) = LET e IN(* ((x,y),c) *)
              LET HA#((Var 0)[1]) IN
              LET HA#(((Var 0)[1], (Var 1)[2])) IN
                     ((Var 0)[1],                  (* sum *)
                      OR#(((Var 0)[2], (Var 1)[2])));(* carry *)
```

The outermost LET is necessary, in case the input expression contains Var constructs. It also avoids duplication of the input circuit by using a fan-out. For example, without this LET, the second half adder in ADD#(Var 0) would incorrectly access the first half adder as input. We easily derive reduceHA and reduceADD using safeOpSemAllTac.

The derived rule reduceHA is listed below.

[3] Note that we don't say *the implementation satisfies the specification.* More below!

```
⊢ o2_ == (case match (T (C (Cons (1,1)),C (Cons (1,1))))
(CoTuple (Cons (n1,1),Cons (n,1))) of
uu => bottomOfConst (Cons (1,1)) |
tt => Cons (1,1) | ff => Cons (2,1))
⊢ o1_ == (case match (B (C (CoTuple (Cons (1,1),Cons (2,1))),
C (CoTuple (Cons (2,1),Cons (1,1)))))
(CoTuple (Cons (n1,1),Cons (n,1))) of
uu => bottomOfConst (Cons (1,1)) |
tt => Cons (1,1) | ff => Cons (2,1))
⊢ (env_ ⊢ circ_ ⇒ (CoTuple (Cons (n1,1),
Cons (n,1)),circ'_) :  TyTuple (Type 1,Type 1))
--------------------
⊢ (env_ ⊢ HA#(circ_) ⇒
(CoTuple (o1_,o2_),HA#(circ'_)) :  TyTuple (Type 1,Type 1))
```

Assuming the following auxiliary functions

```
fun abs (Cons(1,1)) (*hi*) = 1 | abs (Cons(2,1)) (*lo*) = 0;
fun absInv 1 = Cons(1,1) (*hi*) | absInv 0 = Cons(2,1) (*lo*);
val HA_SPEC#(x,y,s,c) = c == (x + y) div 2 ∧ s == (x + y) mod 2;
val ADD_SPEC#(x,y,cin,s,c) = c == (x + y + cin) div 2 ∧
                             s == (x + y + cin) mod 2;
```

we can prove that the half adder satisfies the HA_SPEC specification.

```
⊢ ∀env_,circ_,t_,x,y,a,b.
    (env_ ⊢ circ_ ⇒ (CoTuple (x,y),newcirc_) :
     TyTuple (Type 1,Type 1)) ∧
    (env_ ⊢ HA#(circ_) ⇒ (CoTuple (a,b),HA#(newcirc_)) :
     TyTuple (Type 1,Type 1)) ∧
    (x == hi ∨ x == lo) ∧
    (y == hi ∨ y == lo) → HA_SPEC#(abs x,abs y,abs a,abs b)
```

(We took the liberty of using (*env_* ⊢ *circ_* ⇒ ...) instead of `typeOfExpr` *circ_* ...; *cf.* page 172.) For any input circuit *circ_* delivering (x,y), and half adder's output (a,b), a and b are the sum and carry of x and y respectively. Note that x and y must be well-defined, that is, be `hi` or `lo`. This statement is not very elegant, and this is due to the presence of the abstraction function `abs`. Moreover, any statement about the behaviour of the half adder and its input must involve the static and dynamic semantics.

We would like to use this theorem to change `reduceHA` to use `HA_SPEC` instead of the two explicit computations involving matching. However, this is not possible, because the theorem can be only transformed to

```
⊢ ( ... HA#(circ_) ⇒ ...)
--------------------
⊢ HA_SPEC#(abs x,abs y,abs a,abs b)
```

We want exactly the opposite. To use the specification of a circuit in a derived operational semantics rule we must therefore prove that the circuit and its specification have the same behaviour (under some constraints), rather than the more intuitive 'implementation satisfies specification.' We can derive the rule `reduceHAUseSpec` by proving a bi-implication instead of an implication in `HA_SPEC` above.

```
[4] ⊢ n1 == 1 ∨ n1 == 2
[3] ⊢ n == 1 ∨ n == 2
⊢ out_ == CoTuple (
absInv ((abs (Cons (n1,1)) + abs (Cons (n,1))) mod 2),
absInv ((abs (Cons (n1,1)) + abs (Cons (n,1))) div 2))
⊢ (env_ ⊢ circ_ ⇒ (CoTuple (Cons (n1,1),Cons (n,1)),newcirc_) :
TyTuple (Type 1,Type 1))
--------------------
⊢ (env_ ⊢ HA#(circ_) ⇒ (out_,HA#(newcirc_)) :
TyTuple (Type 1,Type 1))
```

The output from the input circuit *circ_* is used to compute the output *out_* from the half adder directly. The operation `+` at the higher level of abstraction is used rather than the implementation-level matching. Premises `[3]` and `[4]` represent the constraints that the input to the half adder must be either `hi` or `lo`.

To prove this result we considered all combinations of the values x and y are allowed to have. This is verification by exhaustive testing. Although this is necessary for the basic building blocks we use in circuits (*e.g.* basic gates), for larger expressions it is preferable to avoid this method. This should usually be possible by using the specifications of subcircuits and by reasoning about signals as symbolic values. We have not proved any correctness results about `ADD`'s subcircuits, so that we have to use a case analysis. The proof consists of basic rewriting involving `reduce` and the abstraction functions, but is fairly slow. However, for the full adder `ADD` this method becomes prohibitively slow. The embedded operational semantics rules are much faster than the standard rewriting to prove these results. They can only be used in the operational semantics format, however. For example, the correctness statement of `HA` above is in an unsuitable format. The operational semantics rule for `reduceHA` was useful in the derivation of `reduceHAUseSpec`, even though the proof for the latter essentially relied on standard rewriting.

Let us now try to use the `reduceHAUseSpec` to derive a similar rule for `ADD`. It is easy to derive an operational semantics rule for `ADD` which does not unfold the two half adders, but leaves their evaluation as subgoals. These two subgoals can then be rewritten using `reduceHAUseSpec`, resulting in `reduceADDUseHASpec`:

```
[10] ⊢ o2_ == (case match (T (C (Cons (2,1)),C (Cons (2,1))))
(CoTuple (Cons (n1,1),Cons (n,1))) of
uu => bottomOfConst (Cons (2,1)) |
tt => Cons (2,1) | ff => Cons (1,1))
[9] ⊢ n9 == 1 ∨ n9 == 2
⊢ n7 == 1 ∨ n7 == 2
⊢ n10 == 1 ∨ n10 == 2
⊢ n 8 == 1 ∨ n8 == 2
⊢ Cons(n1,1) == AbsInv((Abs(Cons(n9,1)) + Abs(Cons(n7,1))) div 2)
⊢ Cons(n6,1) == AbsInv((Abs(Cons(n9,1)) + Abs(Cons(n7,1))) mod 2)
⊢ Cons(n,1) == AbsInv((Abs(Cons(n10,1)) + Abs(Cons(n8,1))) div 2)
⊢ Cons(n9,1) == AbsInv((Abs(Cons(n10,1)) + Abs(Cons(n8,1))) mod 2)
[1] ⊢ (env_ ⊢ circ_ ⇒
(CoTuple (CoTuple (Cons (n10,1),Cons (n8,1)),Cons (n7,1)),cir'_) :
TyTuple (TyTuple (Type 1,Type 1),Type 1))
--------------------
⊢ (env_ ⊢ ADD#(circ_) ⇒ (CoTuple (Cons (n6,1),o2_),ADD#(circ'_)) :
TyTuple (Type 1,Type 1))
```

Premise [10] represents the OR gate. This rule is not quite satisfactory because it uses HA_SPEC twice, rather than ADD_SPEC once. It turns out that the result that the half adder has the same behaviour as HA_SPEC is not easy to use in the proof that ADD has the same behaviour as ADD_SPEC. The reason is that we have to use standard rewriting to prove the latter, and this is too slow. If, on the other hand, we try to use the embedded operational semantics rules, we cannot combine the premises of reduceADDUseHASpec ([2] to [9] above) to form a single premise involving ADD_SPEC.

It may be that more research into this area sheds some light onto this problem, but currently it seems that this conceptually very powerful hierarchical simulation is less useful than initially thought.

5.2 Hardware Synthesis

We can use the operational semantics rules defined above to synthesise hardware in two different ways, corresponding to top-down and bottom-up synthesis. Another, more powerful method for synthesising circuits in a provably correct way is to use hardware generating functions.

5.2.1 Top-Down Operational Semantics Based Synthesis

We can apply the operational semantics rules to completely unconstrained circuit expressions; *i.e.* flexible meta-variables. Using the usual backward rule applications we refine the circuit top-down. The circuit, and then its subcircuits are being successively refined. A very simple example is the following. All meta-variables, including *circ_*, are flexible. By applying rule `reduceX` to a premise, the circuit expression in that premise becomes an `X` statement.

```
***** Level 3 *****
⊢ (env_ ⊢ circ_ ⇒ (out_,newcirc_) :  t_)
--------------------
⊢ (env_ ⊢ circ_ ⇒ (out_,newcirc_) :  t_)
> apprl reduceTuple;


***** Level 4 *****
⊢ (env_ ⊢ circ2_ ⇒ (o2_,circ2'_) :  t2_)
⊢ (env_ ⊢ circ1_ ⇒ (o1_,circ1'_) :  t1_)
--------------------
⊢ (env_ ⊢ (circ1_,circ2_) ⇒
((o1_,o2_), (circ1'_,circ2'_)) :  TyTuple (t1_,t2_))
```

The application of `reduceTuple` unifies the conclusion of `reduceTuple`

```
(env_ ⊢ (circ1_,circ2_) ⇒ ((o1_,o2_), (circ1'_,circ2'_)):  ...)
```

with the premise of the current goal (*circ_*.) Unification works both ways, so that the current goal may be specialised to allow the application of the rule. Matching would not work here because *circ_* in the premise is not a `Tuple`. The rule `reduceTuple` corresponds to a parallel decomposition of the circuit, `reduceLet` to a sequential decomposition.

```
> apprl reduceLet;


***** Level 5 *****
⊢ (env_ ⊢ circ2_ ⇒ (o2_,circ2'_) :  t2_)
⊢ E t1_1
⊢ o1_1 :  t1_1
⊢ (o1_1 ::  env_ ⊢ circ2_1 ⇒ (o1_,circ2'_1) :  t1_)
⊢ (env_ ⊢ circ1_ ⇒ (o1_1,circ1'_) :  t1_1)
--------------------
⊢ (env_ ⊢ LET circ1_ IN circ2_1,circ2_ ⇒
((o1_,o2_), (LET circ1'_ IN circ2'_1,circ2'_)) :  TyTuple (t1_,t2_))
```

These two steps may be represented diagrammatically as follows. The numbers in the boxes correspond to the subcircuits *circi_j* in the rules.

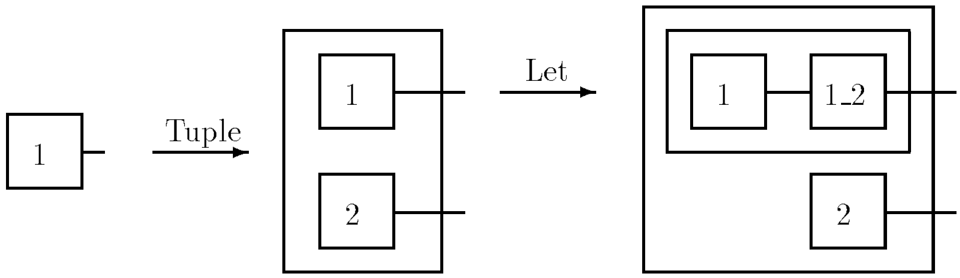

Figure 5–3: Top-down Synthesis using Operational Semantics Rules.

The indexing rules represent a restriction of the outputs. They are more natural to use in a bottom-up synthesis strategy, as we shall see below.

Although this example is very simple, it illustrates the approach. After a number of rules have been applied it makes sense to reflect on the static semantics premises which have been introduced. Often they are duplicated, or may even be eliminated altogether. Unfortunately tactics such as `safeOpSemAllTac` are conservative in their optimisations, especially when flexible variables are present. Using this top-down synthesis we synthesised an adder, but did so with the ready design in hand. In such cases it makes more sense to define an abbreviation for this design and then compute a derived rule `reduceCOMPONENT` using the operational semantics tactics. In both cases a derived rule of the same complexity was arrived at. By using larger building blocks such as AND and OR gates, adders, *etc.* circuits may be synthesised at the block level. It makes more sense to synthesise top-down at a higher level because a hierarchical structure is more apparent there, and the building blocks are larger.

5.2.2 Bottom-Up Operational Semantics Based Synthesis

In LAMBDA backward rule application is the norm. Given a goal we successively break it down until we reach sufficiently small subgoals which can be discharged. That is, we build a proof tree starting at the root from which we work towards the leaves. In forward theorem proving we commence with the leaves and combine proof trees until we have reached the goal we desire. We gave an introduction to forward theorem proving, and how it is supported in LAMBDA in Section 4.1.2. In the operational semantics synthesis example below, we will show only the basic rule which is applied. The actual function which implements the goal stack manipulation[4] takes some extra arguments which are not relevant here.

Using the derived operational semantics rules in a goal directed manner corresponds to a bottom-up synthesis method. For example, forward applying `reduceIndex1` means that we enclose the current circuit with an `Index1` constructor.

[4]`genMergeProofTrees` was briefly discussed in Section 4.1.2.

```
[2] ⊢ E t2_
[1] ⊢ (env_ ⊢ circ_ ⇒ ((o1_,o2_),circ'_) :  TyTuple (t1_,t2_))
--------------------
⊢ (env_ ⊢ circ_[1] ⇒ (o1_, circ'_[1]) :  t1_)
```

In a forward rule application the first premise of `reduceIndex1` is unified with the conclusion of the current goal. This means that the conclusion of the new goal will be the indexed old circuit. The forward application of `reduceIndex1` expects two proof trees. The first tree gives a derivation of *circ_*, the second a proof that the type *t2_* denotes.

The circuit which we build below uses three subgoals, and is synthesised as follows.

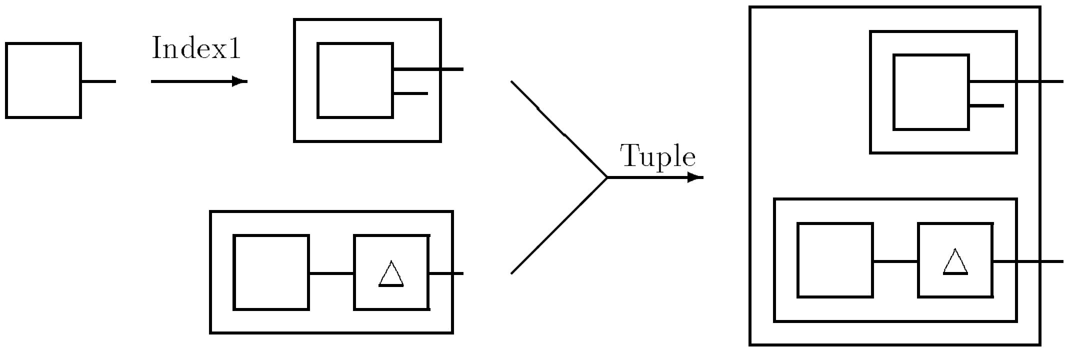

Figure 5–4: Bottom-up Synthesis using Operational Semantics Rules.

The arrow labelled `Index1` combines two proof trees, one `[2]` dealing with the existence of the type *t1_*, and the other `[1]` with the synthesis of the circuit to be indexed. The latter has been omitted from the figure because it deals with the static semantics. Note that in contrast to the top-down synthesis of figure 5–3 boxes are combined (*e.g.* `Tuple`) or added to the outside (*e.g.* `Index1`.) In top-down synthesis boxes are subdivided, or split internally. For this reason, the indexing operators are more natural to use in the bottom-up style than in the top-down style.

We can work with theorems only, as shown below. However, it is more useful to have general input circuits and states, as we saw in Section 5.1. This is why we use the following trivial proof of a circuit which is delayed. Both the circuit and the state in the delay are not specialised.

```
> pushRule opsempe reduceDelay;


***** Level 1 *****
⊢ initial_ :  t_
⊢ (env_ ⊢ circ_ ⇒ (out_,circ'_) :  t_)
--------------------
⊢ (env_ ⊢ DELAY (initial_,circ_) ⇒
(initial_,DELAY (out_,circ'_)) :  t_)
```

The following subproof shows the existence of the second type of the `Index1` rule. Note that there are now two proof trees on the goal stack.

```
> pushRule opsempe reduce0;


***** Level 1 *****
--------------------
⊢ E 0
> forwardApprl 1 reduceSn;


***** Level 2 *****
--------------------
⊢ E 1
> forwardApprl 1 reduceType;


***** Level 3 *****
--------------------
⊢ E (Type 1)
```

`trivialOpSemRule` is the trivial operational semantics rule. It will represent the circuit which provides an input to the `Index1`.

```
> pushGoal opsempe trivialOpSemRule;


***** Level 1 *****
⊢ (env_ ⊢ circ_ ⇒ (out_,newcirc_) :  t_)
--------------------
⊢ (env_ ⊢ circ_ ⇒ (out_,newcirc_) :  t_)
```

There are now three subproofs on the goal stack. From the bottom to the top of the stack these are: the delay derivation, the $\vdash$ `E (Type 1)` derivation, and the trivial operational semantics rule. The forward rule application of `reduceIndex1f` merges the top two subproofs into one proof. For technical reasons `reduceIndex1f` is slightly different from `reduceIndex1` which is used in backward theorem proving. See appendix C for details.

Thus the derivation of `E(Type 1)` is unified with premise `[2]` of `reduceIndex1`, and the (trivial) derivation of rule `trivialOpSemRule` is unified with premise `[1]`. The overall effect is that the type *t2_* is unified with `Type 1`, and *circ_* remains unchanged.

```
> ... (* forward apply *) reduceIndex1f;


***** Level 4 *****
⊢ (env_ ⊢ circ_ ⇒ ((o1_,o2_),newcirc_) :  TyTuple (t1_,Type 1))
--------------------
⊢ (env_ ⊢ circ_[1] ⇒ (o1_,newcirc_[1]) :  t1_)
[Merged top two proof trees; StackLevel=2]
```

Similarly, the application of `reduceTuple` combines the top two subproofs. The current goal (at level 4, above) is unified with the first premise of `reduceTuple` and the (trivial) derivation of `reduceDelay` is unified with the second component of the tuple.

```
> ... (* forward apply *) reduceTuple;


***** Level 2 *****
⊢ (env_ ⊢ circ_1 ⇒ ((o1_,o2_),newcirc_) :  TyTuple (t1_,Type 1))
⊢ initial_ :  t_
⊢ (env_ ⊢ circ_ ⇒ (out_,circ'_) :  t_)
--------------------
⊢ (env_ ⊢ (circ_1[1],DELAY (initial_,circ_)) ⇒
((o1_,initial_),(newcirc_[1],DELAY (out_,circ'_))) :  TyTuple (t1_,t_))
[Merged top two proof trees; StackLevel=1]
```

This method of synthesis is quite hard to use because, as with general goal-directed theorem proving, one must anticipate what subgoals will be needed later in a proof. Moreover, their format must be quite precise. In this respect, having a rigid format such as the one used by the operational semantics rules is an advantage.

5.2.3 Hardware Synthesis Functions

One of the strengths of the embedding approach used here is that we can manipulate circuit expressions just like any other term in the proof system. This allows us to write functions operating on and delivering circuits.

We will define an N bit adder generating function which is parametrised on a full adder subcomponent. The arguments of the functions are nested binary tuples:

```
onebitadder:  ((x,y),c) -> (s,c)
nadd:  (((x_{N+1},(..,x_0)),(y_{N+1},(..,y_0))),c_0) -> ((s_{N+1},(..,s_0)),c)
```

```
fun nadd onebitadder (S 0) x = onebitadder x |
    nadd onebitadder (S (S n)) x =
         LET x IN    (* (((x_{N+1},x̄),(y_{N+1},ȳ)),c_0) *)
         LET nadd onebitadder (S n)
             (((Var 0)[1][1][2], (Var 0)[1][2][2]), (Var 0)[2]) IN
         LET onebitadder (((Var 1)[1][1][1], (Var 1)[1][2][1]),
                          (Var 0)[2]) IN
             (((Var 0)[1], (Var 1)[1]),     (* sum *)
              (Var 0)[2])                   (* carry *);
```

`nadd: (`*expr* `->` *expr*`) ->` *natural* `->` *expr* `->` *expr* is a partial function: there is no such a thing as a zero bit adder. A one bit adder with input $((x_0,\ y_0),\ c_0)$ uses the full adder component. An $N+1$ bit adder with input $(((x_N,\ \overline{x}),\ (y_N,\ \overline{y})),\ c_0)$ uses an N bit adder with input $((\overline{x},\ \overline{y}),\ c_0)$ connected to a full adder with input $((x_N, y_N), c_N)$. As with the `ADD` circuit, virtually all of the complexity is due to the composition of intermediate wires.

We can use this function definition in conjunction with the embedded operational semantics rules as follows. The derived rule `reduceNADD1` just unfolds the `nadd` definition to evaluate the full adder. Note that the result circuit must be identical to the circuit we evaluate. This means that the adder is not allowed to have any state. (This is a design decision, not an inherent restriction.)

$\vdash$ (*env_* $\vdash$ *add_ circ_* $\Rightarrow$ (*out_*,*add_ circ'_*) : *t_*)

$\vdash$ (*env_* $\vdash$ `nadd` *add_* `1` *circ_* $\Rightarrow$ (*out_*,`nadd` *add_* `1` *circ'_*) : *t_*)

The derived rule `reduceNADDSSn`, dealing with $N+2$ word size, is more involved. Premise one evaluates the input circuit, premise two the $N+1$ bit adder, and premise three the full adder. The remaining premises deal with the static semantics. We see that the output of the $N+1$ bit adder is a tuple `CoTuple (`*o2_2*`, Cons (`*n3*`,`*m2*`))`. Comparing this to the definition of `nadd` we see that *o2_2* represents the partial sum $(s_N,(..,s_0))$, and `Cons` $(n3,m2)$ the carry c_{N+1}. Decoding the inputs of the final $N+2$nd bit adder *add_*, we see that its input carry `(Var 0)[2]` accesses the output carry `Cons`$(n3,m2)$ from the $N+1$ bit adder, as expected. The final result of the $N+2$ bit adder consists of (i) the concatenation of the sum bit of *add_* (`Cons`$(n2,m1)$) with the partial sum *o2_2*; and (ii) the carry bit `Cons`$(n1,m)$ of *add_*.

```
⊢ E t1_
⊢ o1_ :  t1_
⊢ E (Type m2)
⊢ E t1_3
⊢ o2_2 :  t1_3
[3] ⊢ (CoTuple (o2_2,Cons (n3,m2)) ::  o1_ ::  env_ ⊢
add_ (((Var 1)[1][1][1], (Var 1)[1][2][1]), (Var 0)[2]) ⇒
(CoTuple (Cons (n2,m1),Cons (n1,m)),
add_ (((Var 1)[1][1][1], (Var 1)[1][2][1]), (Var 0)[2])) :
TyTuple (Type m1,Type m))
[2] ⊢ (o1_ ::  env_ ⊢ nadd add_ (S n)
((((Var 0)[1][1][2], (Var 0)[1][2][2]), (Var 0)[2])) ⇒
(CoTuple (o2_2,Cons (n3,m2)),nadd add_ (S n)
((((Var 0)[1][1][2], (Var 0)[1][2][2]), (Var 0)[2]))) :
TyTuple (t1_3,Type m2))
[1] ⊢ (env_ ⊢ circ_ ⇒ (o1_,circ'_) :  t1_)
--------------------
⊢ (env_ ⊢ nadd add_ (S (S n)) circ_ ⇒
(CoTuple (CoTuple (Cons (n2,m1),o2_2),Cons (n1,m)),
nadd add_ (S (S n)) circ'_) :
TyTuple (TyTuple (Type m1,t1_3),Type m))
```

A four bit adder has been simulated, with `ADD` as the subcomponent. For example, binary $1010 + 1101 + 1 = 11000$, that is, a sum of 1000 and a high carry:

```
> applyTac (safeOpSemAllTac'[reduceNADD1bit']);


***** Level 4 *****
--------------------
⊢ ([(((hi,(lo,(hi,lo))), (hi,(hi,(lo,hi)))), hi)], env_ ⊢
nadd (fn e => ADD#(e)) 4 (Var 0) ⇒
([((hi,(lo,(lo,lo))), hi)], nadd (fn e => ADD#(e)) 4 (Var 0)))
```

Note that `ADD` is a meta-level syntactic function, and must therefore be converted into an object-level function, using `(fn e => ADD#(e))`.

While using the circuit generating function in this manner is useful to contain the complexity of circuits, there is a more important aspect to their use. We can verify the correctness of a circuit generator, rather than verify the correctness of individual outputs. This means that for any circuit which correctly implements a one bit adder, and for any N the output of `nadd` is guaranteed to be a correct N bit adder.

Research into Formal Synthesis

Formal circuit generators such as `nadd` were introduced by Brock *et al.* in [22,23] for use in the Boyer-Moore theorem prover. Hardware generators have also been formally verified in NUPRL [9] and HOL [38]. Chin's approach [38] is distinctive because no HDL is used. Synthesis functions output relational hardware descriptions in higher-order logic instead (*cf.* Section 2.1.)

Hanna's formal synthesis methodology [86] is a top-down design approach. *techniques*, analogous to tactics, are used to split a goal into subgoals and a justification why this step is justified. The specification is reduced to simpler parts and a circuit is designed at the same time. Our operational semantics rules are used for top-down synthesis in a more limited manner because no specification is associated with a design. Synthesised circuits must be verified after they have been completed. The formal synthesis methodology could probably be adapted to work with picoELLA in LAMBDA, if rules are used as techniques.

The DIALOG synthesis system [63,62,59], which is integrated with LAMBDA, is a schematic editor. It allows designers to synthesise circuits formally by instantiating components and connecting them using a graphical interface. Components and their interconnections have an underlying logical representation which is used to prove correct the current specification. The design is proof driven in the sense that the designer will decide which components to use, and where to connect them depending on the current outstanding proof obligations. Changes to the design update the state of the proof. At no point is the user obliged to type proof system commands. Currently the underlying representation of circuits uses a relational style, but in principle it would be possible to use an embedded HDL. Proof obligations will become more complex, however, due to the fact that the behaviour of a component must be derived via a semantics. A library of standard components and previously derived behaviours could be provided to ease this process. Such a system would hide most of the proof system from a designer; after supplying a formal specification only HDL descriptions and the corresponding schematic representation are observed.

5.3 Transformations on Circuits

Circuit transformations are often used to optimise hardware designs. A circuit may be replaced by another which has the same behaviour but which is more desirable in some other way. Examples of such properties are speed of operation, and silicon area required. Note that the differences in these properties are not derivable from the semantics because otherwise the circuits would not be behaviourally equivalent. To show how one could proceed to use this idea in practice consider the following definition of behavioural equivalence.

The approach we have taken here is to provide a mapping from one circuit to its preferred form. To ensure that this optimisation function maps well-formed circuits to well-formed circuits we test this for each invocation. (An alternative is to require this as an invariant on the function.) The predicate `WF` implements this.

```
val WF#(l,e,fe) =
    ∃t.  typeOfExpr (map typeOfConst l) e == (t,true) ∧
         typeOfExpr (map typeOfConst l) (fe e) == (t,true);
```

The predicate `BEHEQ` asserts that the original circuit and its transformed form have the same outputs. Being the same is not equality in this case, because the output from an evaluation of a circuit results in a constant output and a new circuit description. Only the constant outputs are the same; the new circuits stand in the same relation as the input circuits. That is, input circuit e is transformed to *fe* e, and the output circuit f must therefore also be transformed by *fe*. This may be expressed as follows.

```
val BEHEQ#(l,e,fe) =
    (fn (c,e) => (c,fe e)) (reduce l e) == reduce l (fe e);
```

These two abbreviations may be combined conveniently as follows.

```
val BEQ#(l,e,fe) = WF#(l,e,fe) → BEHEQ#(l,e,fe);
```

Note that there is a restriction on the transformation function due to the formulation of behavioural equality in this manner. The function *fe: expr* `->` *expr* maps a circuit, including its embedded state, to another circuit, possibly with another embedded state. Thus there must be a functional relationship between the two circuits at every point in time. We will encounter an example below where there is no such simple relation. We have to generalise our notion of behavioural equivalence there.

To illustrate how such transformations may be proved correct consider the following transformation function.

```
(fn (Delay (CoTuple (c,d), Tuple (e,f))) =>
    Tuple (Delay (c,e), Delay (d,f)))
```

It converts a delayed `Tuple`, or parallel composition, into the parallel composition of the delayed subcircuits. The following theorem may be proved easily.

```
⊢ ∀l,c,e.  BEQ#(l, Delay (c,e),
                (fn (Delay (CoTuple (c,d), Tuple (e,f))) =>
                    Tuple (Delay (c,e), Delay (d,f))))
```

An induction on c ensures that c is a constant tuple. Similarly, an induction on e forces it to be a tuple. The proof then unfolds the abbreviations and moves all antecedents of the implications to the hypothesis lists. The well-formedness condition on the delayed tuple can then be used to show that the subexpressions of the conclusion are well-formed. Rewriting using `reduce` and `typeOfExpr` then yields the right hand side.

In a similar manner we proved a number of transformations to remove the indexing operators from a circuit. We only show the `Index1` rules, the `Index2` rules are very similar.

```
⊢ ∀l,c.  BEQ#(l, Index1 (Const c),
              (fn (Index1 (Const (CoTuple (c,d)))) => Const d))
```

```
⊢ ∀l,e,f.  BEQ#(l, Index1 (Tuple (e,f)),
                (fn (Index1 (Tuple (e,f))) => e))
```

```
⊢ ∀l,c,e.  BEQ#(l, Index1 (Delay (c,e)),
                (fn (Index1 (Delay (CoTuple (c,d),e))) =>
                    Delay (c,Index1 e)))
```

```
⊢ ∀l,e,f.  BEQ#(l, Index1 (Let (e,f)),
                (fn (Index1 (Let (e,f))) => Let (e,Index1 f)))
```

```
⊢ ∀l,c,e,f.  BEQ#(l, Index1 (LetRec (c,e,f)),
                  (fn (Index1 (LetRec (c,e,f))) =>
                     LetRec (c,e,Index1 f)))
```

```
⊢ ∀l,e,f,g,ch.  BEQ#(l, Index1 (If (e,f,g,ch)),
                    (fn (Index1 (If (e,f,g,ch))) =>
                       If (e,Index1 f,Index1 g,ch)))
```

We cannot prove a useful relationship between `Var` constructors and the indexing operators. There are two reasons for this. The first is that when we arrive at the evaluation of a `(Var 0)[1]` say, we don't know anything about the defining expression. To try and connect the definition and its use is difficult due to the possible presence of nested `Let` statements. We cannot replace `(Let (`e`,`f` (Index1 (Var 0))))`, where f is a function from expressions to expressions, by `(Let (Index1 `e`,`f` (Var 0)))` because the `Var 0` may be accessing a more deeply nested `Let`. The second reason is that the defining expression may be used in more than one way. Consider `Let (`e`, Tuple (Index2 (Var 0), Index1 (Var 0)))`, which interchanges the outputs of e.

We proved the transformations in the form of theorems. While this is conceptually clearest, they are more useful in the following format:

```
⊢ BEHEQ#(l,e,fe)
... ⊢ P#((fn (c,e) => (c,fe e)) (reduce l (fe e)))
--------------------
... ⊢ P#(reduce l e)
```

That is, we can replace an evaluation of e by an evaluation of fe e, in any context `P`. However, we must also prove that e and fe e are behaviourally equivalent. The rule which justifies pushing a delay through a tuple is an example of a similar format.

```
1:   (l ⊢ DELAY ((c,d),(e,f)):  t)
⊢ P#((fn (c1,e1) => (c1, (fn (Delay (CoTuple (c,d),Tuple (e,f))) =>
Tuple (Delay (c,e),Delay (d,f))) e1))
(reduce l (Delay (CoTuple (c,d),Tuple (e,f)))))
--------------------
1:   (l ⊢ (DELAY(c,e),DELAY(d,f)):  t)
⊢ P#(reduce l ((DELAY (c,e),DELAY (d,f))))
```

(Recall that ($l \vdash$ e: t) indicates that e is well-typed.

Unfortunately, this rule is the wrong way round: we replace the desirable circuit *fe e*, or in this case (DELAY (c,e),DELAY (d,f)), by the less desirable e, in this case (DELAY ((c,d),(e,f))). It is not hard to prove the reverse rule for this particular example, but the optimisations involving the indexing operators are problematic. The reason is that the transformation function forgets part of the embedded state. In the rule which pushes `Index1` through a `Delay`, `CoTuple` (c, d) is projected to c. We can convert this rule to the same format as the rule above but this is not very useful because it introduces an irrelevant element in the state. We cannot, however, define an inverse *fe* function to prove the inverse rule. This is a simple example of a non-functional relationship.

All optimisations we have encountered so far have been constant over time. That is, the transformation function has not depended either explicitly on time, or implicitly, by being dependent on the state of a circuit. Our transformations involving delays merely redistributed the state, they did not depend on the particular values in the delays. Converting these to apply over more than one time step, that is, involving `reduceSeq` is not very useful, because the circuits then become top-level circuits rather than subcircuits. It is preferable to work with circuit fragments and `reduce` as these can be replaced in any context, regardless of time.

A more general correctness condition may be defined thus. Given an environment l, an initial well-formed circuit e, and a transformation function *fe: time* `->` *expr* `->` *expr*, we can conclude that the circuit e and its transformed form *fe t e* produce the same output for all times t.

```
val CORRECT#(l,e,fe) =
(∃s.  typeOfExpr (map typeOfConst l) e == (s,true)) →
∀f.  (f 0 == e ∧
      ∀t.  f (S t) == (fn (_,e) => e) (reduce l (f t)) ∧
           (shapeEq e (f t) == true ∧
            shapeEq (f t) (f (S t)) == true)) →
     ∀t.  (∃s.  typeOfExpr (map typeOfConst l) (f t) == (s,true) ∧
                typeOfExpr (map typeOfConst l) (fe t (f t)) ==
                                                    (s,true)) ∧
          (fn (c,e) => (c,fe (S t) e)) (reduce l (f t)) ==
                                     reduce l (fe t (f t));
```

The implicitly defined function *f: natural* `->` *expr* maps a time to the circuit description at that time. The invariants involving `shapeEq` state that the circuit, at any time, has the same shape as the original circuit e, and also as the circuit at the next time step. This information is needed to be able to use the definition of *fe*, which needs to know the structure of the expression it is applied to. The well-typedness assertions in the conclusion serve as additional invariants with a similar purpose. Nothing can be done with a circuit description unless we know that it is well-typed. Note that *fe* depends on time t and in the conclusion *fe t* is applied to a circuit at time t, and *fe* `(S` t`)` is applied to the new circuit description (at time `S` t.)

As an example of a transformation function which uses the full power of this correctness statement reconsider the two parity checkers of Section 5.1.3. The complexity of the `HOLPC` parity checker (figure 5–1 on page 186) was mostly due to the use of `lo` initial value in the delays. The `PC` parity checker was almost trivial due to the use of a `hi` initial value in its delay. We may regard this difference as an abstraction function. At one level of abstraction the initial values in delays may be arbitrary, but at a lower level of abstraction they must be equal to a particular value. We want to transform an initial description `DELAY` (b,e) to a circuit using only delays with initial value a at time zero. This transformation function may be defined as follows.

```
fun fe a b 0 (Delay (x,e)) =
            If (Delay(a,Const b),Const b,Delay (a,e),C a) |
    fe a b (S t) (Delay (x,e)) =
            If(Delay(b,Const b),Const b,Delay (x,e),C a);
```

The function `fe` maps `Delay` (x,e) to `Delay` (a,e) at time `0`, but to itself thereafter. This is the time dependency of `fe`. For a permitted initial value a, and an original initial value b we create a multiplexor which selects b at time zero, and the delayed original circuit e at later times. This is accomplished by using a circuit `DELAY` (a,b) which outputs a at time zero, and b from then on.

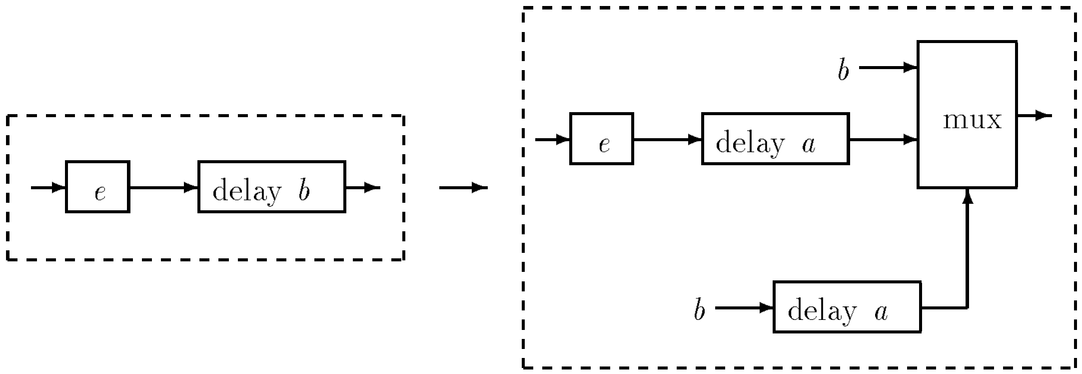

Figure 5–5: Initial Value of Delay Transformation.

We prove the following correctness result. The time dependence of `fe` may be seen from the state of the first delay, which is a at time zero, and b otherwise. `fe` is also an injection; the result circuit `HOLPC` contains a larger state (more information) than the initial circuit `PC`.

```
⊢ ∀l,a,b,e.
   (typeOfConst a == typeOfConst b ∧ match (C a) b == ff) →
   CORRECT#(l,Delay(b,e),fe a b)
```

This is a straightforward theorem, apart from the second antecedent. It states that the multiplexor in the design must select its `ELSE` branch from time zero onwards. *i.e.* a and b must not match, when a is interpreted as a chooser. This rather technical side condition was initially omitted from the theorem, but turned out to be necessary. A more intuitive statement would be that a and b must be not related in the data ordering `cle`. Some valid combinations of `(`a`,`b`)` are `(lo,hi)`, `(hi,lo)`, `((lo,lo), (hi,bit))`, *etc.* `fe lo hi` would be the transformation to use to (partially) convert `PC` to `HOLPC`.

We could have deduced a slightly more liberal precondition, but it would have relied upon a proof by contradiction. In other words, in certain cases the output of the transformed circuit could have been incorrect, but the weaker precondition would have been inconsistent. In retrospect it is clear that in this case we would rather have a stronger precondition and not use the *ex falso* rule. However, we *derived* the constraint on a and b because it was not clear at the outset what a good precondition would be. When it came to instantiating the constraint during the course of the proof it was not obvious why we should not use the weakest possible precondition, instead of the more restrictive `match (C` a`)` b `== ff`. During the proof we arrived at the two following subgoals.

2:	`match (C` *a'*`)` *b'* `== uu` ⊢ *a1'* `== bottomOfConst` *b'*
1:	`match (C` *a'*`)` *b'* `== tt` ⊢ *a1'* `==` *b'*

The term *a1'* is the output at this time step. Any relation between *a'* and *b'* which ensures that these two goals are satisfied is a valid precondition. If we do not want to use a proof by contradiction, in the second premise the output at this time step must be equal to bottom. In the first premise it must be the same as the output at time zero, *i.e.* *b'*. In both cases time is a free variable and this means that the circuit always outputs undefined (in the former case) or *b'* (in the latter case.) This would constrain the circuit e, to be transformed, quite severely! By including `match (C` a`)` b `== ff` as an antecedent `1` and `2` do not arise, and no proof by contradiction is necessary.

This particular problem is, of course, an instance of the *false implies everything* problem [185,179,13].

Note that at least two distinct non-bottom values are needed in the value domain for the theorem to hold. If we have at least three distinct non-bottom values we could prove a result which does not constrain the relation between a and b. `fe` could be redefined thus.

```
fun fe' a b c 0 (Delay (x,e)) =
                If (Delay(a,Const c),Const b,Delay (a,e),C a) |
    fe' a b c (S t) (Delay (x,e)) =
                If(Delay(c,Const c),Const b,Delay (x,e),C a);
```

The constraint generated by this definition, in the amended statement of the theorem, would be `match (C` a`)` c `== ff`. This relaxation allows b to be bottom, where previously this was not allowed.

A more realistic example of a transformation function is `replicate` which removes all `LET` statements and makes all recursive `LET` statements as simple as possible. Of course, the circuit may become much larger as a result of this expansion. Unfortunately we have not had the opportunity to verify this function. It could be used in conjunction with a function which computes the fan-out of defining expressions. If the fan-out is too large, some of the references to the defining circuit (*i.e.* `Var` constructors) could be replaced by the defining expression. The expression environment l links the de Bruijn encoded variables with their defining expressions.

```
fun replicate l (Const c) = Const c |
    replicate l (Tuple (e,f)) = Tuple (replicate l e,replicate l f) |
    replicate l (Let (e,f)) = (replicate (replicate l e::l) f) |
    replicate (h::t) (Var 0) = h |
    replicate (h::t) (Var (S n)) = replicate t (Var n) |
    replicate l (Delay (c,e)) = Delay (c,replicate l e) |
    replicate l (If (e,f,g,ch)) = If (replicate l e, replicate l f,
                                      replicate l g, ch) |
    replicate l (Index1 e) = Index1 (replicate l e) |
    replicate l (Index2 e) = Index2 (replicate l e) |
    replicate l (LetRec (c,e,f)) =
       replicate (LetRec (c, replicate (Var 0::l) e,Var 0)::l) f;
```

Research into Formal Transformations and Optimisations

In [103, Chapter 6] Johnson shows how transformations can be proved to be behaviour preserving using a formal semantics of a stream-based language DAISY. This is basically the approach used in this section. An algebraic approach to transformational design has been presented by Johnson and Zhu in [104,188]. Transformations on circuits have been used to optimise regular designs [115]. All the above research has been equational in nature. Busch has extended this to implicational transformations [30,31].

Our approach to correct transformations and optimisations is powerful, but hard to use. The general correctness statement `CORRECT` is quite complex because the transformation is dependent on the circuit which is transformed, its embedded state, and the time at which it is applied. It may well be better to use only simple transformations, and without an explicit transformation function *fe*.

5.4 Discussion

In this section we reflect on our experience with the embedded operational semantics.

Proofs and Invariants

Recall the statement of correctness of the `PC` parity checker of page 190:

```
⊢ ∀l,e,input.
    (∀t.  reduce l (e t) == (input t,e (S t)) ∧
          (input t == hi ∨ input t == lo)) →
                 ∀t.  reduce l (PC#(state input t,e t)) ==
                        (state input t, PC#(state input (S t),e (S t)))
```

We cannot prove this statement as it stands. In all reasoning about picoELLA circuits containing state, an invariant expressing the desired property about the circuit must be provided. In this case we prove the following lemma first:

```
⊢ ∀l,e,input.
    (∀t.  reduce l (e t) == (input t,e (S t)) ∧
          (input t == hi ∨ input t == lo)) →
          ∀t.  reduce l (PC#(state input t,e t)) ==
                 (state input t, PC#(state input (S t),e (S t))) ∧
                 (state input (S t) == hi ∨ state input (S t) == lo)
                 (* Need extra invariant on parity's state.  *)
```

The extra part of the conclusion states that the state is always `hi` or `lo`.

Another feature, which we also encountered in the definition of `CORRECT` (page 219), is the time-to-expression mapping *e: natural* `->` *expr*. As circuit expressions which contain a state have a different description at every time step, it is necessary to introduce a function which expresses this. This forces proofs to include an induction over time. We can prove the result for the initial time `0`, where we know what the circuit looks like. For the inductive case we can use the induction hypothesis. It is often not clear exactly what the statement of the theorem is when these modifications have to made. For example, the above theorem is more intuitively stated as follows.

$$\begin{array}{l}
\vdash \forall t, l, e, input. \\
\quad \texttt{reduce}\ l\ (e\ t)\ \texttt{==}\ (input\ t, e\ (\texttt{S}\ t))\ \wedge \\
\quad (input\ t\ \texttt{==}\ \texttt{hi} \vee input\ t\ \texttt{==}\ \texttt{lo}) \rightarrow \\
\qquad\qquad \texttt{reduce}\ l\ (\texttt{PC\#(state}\ input\ t, e\ t))\ \texttt{==} \\
\qquad\qquad (\texttt{state}\ input\ t,\ \texttt{PC\#(state}\ input\ (\texttt{S}\ t), e\ (\texttt{S}\ t)))\ \wedge \\
\qquad\qquad (\texttt{state}\ input\ (\texttt{S}\ t)\ \texttt{==}\ \texttt{hi} \vee \texttt{state}\ input\ (\texttt{S}\ t)\ \texttt{==}\ \texttt{lo})
\end{array}$$

However, this statement cannot be proved because in the inductive case of the induction on t, `reduce` l (e t) is needed to discharge the antecedent of the induction hypothesis. We only have the output of `reduce` l (e (`S` t)) available. The first statement of the theorem which was correct was the following.

$$\begin{array}{ll}
\forall t, l, e, input. & \texttt{reduce}\ l\ (e\ t)\ \texttt{==}\ (input\ t, e\ (\texttt{S}\ t))\ \wedge \\
 & (input\ t\ \texttt{==}\ \texttt{hi} \vee input\ t\ \texttt{==}\ \texttt{lo}) \\
\vdash \forall t, l, e, input. & \texttt{reduce}\ l\ (\texttt{PC\#(state}\ input\ t, e\ t))\ \texttt{==} \\
 & (\texttt{state}\ input\ t,\ \texttt{PC\#(state}\ input\ (\texttt{S}\ t), e\ (\texttt{S}\ t)))\ \wedge \\
 & (\texttt{state}\ input\ (\texttt{S}\ t)\ \texttt{==}\ \texttt{hi} \vee \texttt{state}\ input\ (\texttt{S}\ t)\ \texttt{==}\ \texttt{lo})
\end{array}$$

This is a rather heavy-handed version, with an excessive amount of generality, as a response to the problem described above. It works because we can instantiate the hypothesis with the appropriate t, l, e, and *input* without having to worry about any implications in induction hypotheses. Some reshuffling of quantifiers resulted in the nicer correctness statement given at the start of this section.

In addition to the introduction of explicit time dependent circuit expressions, the variable *input* is needed to describe the output of the input circuit. The result type *const* `*` *expr* of the dynamic semantics function `reduce` means that we have to name explicitly the output circuit and the constant output, or use projections to obtain them. In both cases, the resulting expressions are time dependent. The decision to omit the picoELLA `INPUT` construct results in using (the de Bruijn encoded equivalent of) free variables to represent inputs to a circuit. We can either insert input values directly into the environment, *e.g.*

```
P#(reduce (input t::l) (PC#(Var 0)))
```

or use an unspecified input circuit e

```
reduce l (e t) == (input t,e (S t)) ... ⊢ P#(reduce l (PC#(e t)))
```

The former limits the parity checker to be used inside a `LET` statement to access the input value through the `Var 0`. The latter form is more verbose and introduces more variables but allows `PC` to be used in any context. Moreover, `PC` must save its input expression and use a fan-out to prevent incorrect environment accesses which e may contain (*cf.* `PC`'s definition on page 188.) Either way, the result is tedious to work with.

Apart from invariants on the state of circuits, invariants on the shape may be needed. An example of this is the definition of `CORRECT` on page 219. Although the monotonicity theorem of `reduce` (Section 4.3.2) tells us that the shape of circuits does not change over time, this is expressed in terms of one time step to the next. In the case of `CORRECT` we also want to know that the circuit is shape equal to the initial circuit because the transformation functions *fe* are defined on the structure of the terms they operate on. We are only given the shape of the initial circuit. For similar reasons we need to know that the results of reductions are well-typed. Again, `reduceMonotoneT` may be used for this, but it cannot be used to discharge well-typedness antecedents of induction hypotheses for similar reasons as above.

Problems with picoELLA

Specific problems of picoELLA include the treatment of undefined, or bottom values. This was noted also in the HOL embedding of ELLA, which led to the decision to abandon the original semantics which used lifting to deal with the undefined value [17] in favour of a non-conforming semantics [18, Section 7.8]. (We discussed the latter semantics in Section 3.3.6, and the embedding of undefined values in

Section 4.2.4.) It is the presence of the don't know value as an extra constant in the value domain which makes the matching function `match` more sophisticated than structural equality on constants. Undefined values cannot be encoded by the iota operator in LAMBDA, or the corresponding Hilbert operator in HOL.

In all verification examples we performed we excluded partially defined values from the legal inputs (*e.g.* the parity checker `PC`, above, and the correctness of the half adder on page 201.) This ensures that the abstraction functions do not have to map the bottom value to a corresponding undefined value of the more abstract value domain. Due to the restrictions on the iota operator in LAMBDA, it would be very annoying to implement this. For example, if tuples of bits are mapped to natural numbers we would like to map partially defined constants to an iota term.

```
fun abs (Cons (0,t)) = (ιx:  natural.  x == 0 ∨...∨ x == maxint) |...
```

This is currently not allowed in LAMBDA, and we would have to encode the undefinedness as a separate data type (*e.g.* by lifting [17].) As discussed earlier in Section 5.1.4, circuit without delayless feedback loops cannot give rise to undefined values unless undefined values are input. Thus abstraction functions need not deal with undefined values for intermediate values, outputs, or embedded state. In theory, this allows us to transform premises such as

```
case match hi c of uu => bottomOfConst lo | tt => lo | ff => hi
```

into something like `if c = hi then lo else hi` or `absInv (not (abs c))`. In practice, neither turned out to be easy to reason about.

Another problem of picoELLA is that it has no real facilities to deal with data abstraction within the language. The use of binary tuples and enumerated types alone, make data types such as bitvectors tedious to work with (*cf.* `nadd` on page 211.) ELLA contains finite range integers [151, Section 4.4.1.2] on which a number of built-in operators work. In theory, ELLA integers are just another enumerated type. In practice, however, the built-in operators provide a crucial functionality. For example, if we define the ELLA integer type `address`, and a function to add two addresses

```
TYPE address = NEW addr(0..255).
FN ADDADDRESS = (address:  i j) -> address:
CASE (i,j) OF
(addr/0,address):  j,
(addr/1,addr/1)|(addr/2,addr/0):  addr/2,
...
(addr/1,addr/254)|...|(addr/255,addr/0):  addr/255
ELSE ?address ESAC.
```

The same approach has to be taken in picoELLA, and this is clearly not feasible. In ELLA the built-in operator `PLUS_US` may be used instead:

```
FN ADDADDRESS = (address:  i j) -> address:  BIOP PLUS_US.
```

The semantics of `BIOP` statements was given in [92]. The definition of `PLUS_US` shows that the addresses are converted to unsigned integers. These are added and converted back to addresses.[5] Overflow results in the bottom value being returned by the `BIOP`. This is a rather *ad hoc* solution, which has been used for integers and reals. This conversion can be conveniently expressed in the LAMBDA embedding of picoELLA:

```
fun enumToNat (Cons (S i,j)) = i;
```

We have ignored any typing constraints here. Note also that `enumToNat` is undefined for bottom values. The `PLUS_US` operator can be coded as follows:

[5] To be precise, this was the approach taken for ELLA version 3.0 `ARITH` statements [151, Section 6.4.1.1] which were superseded by the `BIOP` constructs of later versions [136, Section 2.3]. All arguments to `BIOP` statements are converted to bit strings first [92,45], which we have omitted above.

```
fun plus n (CoTuple (Cons(0,s),Cons(j,t))) = Cons(0,s) |
    plus n (CoTuple (Cons(j,s),Cons(0,t))) = Cons(0,s) |
    plus n (CoTuple (Cons(S i,s),Cons(S j,t))) =
           if n < i + j then Cons(0,s) else (Cons(S i + j,s));
```

It is preferable to incorporate `enumToNat` in `plus` because it can deal with undefined values more easily. The function `plus` is strict in the sense that bottom values and any overflow in the course of the addition will produce an undefined output. The first argument n indicates the maximum element of the ELLA integer range. It is supplied explicitly because we do not know how many elements a type has (see Section 4.2.1.) The semantics of picoELLA can be extended to use the new built-in operators as follows.

```
datatype biop = Plus of natural | ...;
datatype expr = Biop of biop | ...;
fun reduce l (Biop (Plus n)) = plus n (reduce l) (Var 0) | ...;
```

Although it is possible to add integers and other high-level data types to picoELLA in this manner, we would prefer a more aesthetically pleasing solution.

Another problem, which is not unique to the picoELLA approach, is that the state of a circuit is visible at all stages. There is no mechanism for hiding the state. Thus, abbreviations for circuits with memory such as `PC` and `HOLPC` are parametric on the state, while rules for `ADD` and `nadd` specifically exclude any embedded state. This is due to the lack of operators for hiding the state in delays. In relational hardware descriptions the logical existential operator $\exists$ may be used for this purpose. An abbreviation or circuit description must always be parametric on any embedded state. Abstraction functions and state invariants, such as `state` for `PC`, can relieve this problem, but it remains a problem nonetheless.

5.5 Conclusions

In this chapter we presented a number of very small examples to illustrate the various ways in which an embedded operational semantics could be used. From a practical point of view, the use of a semantics to derive the behaviour of circuits tends to be quite tedious, especially because rewriting of `reduce` tends to be slow. Anything larger than a one bit adder is not practical to verify using rewriting. For example, proving `PC` correct takes nearly three hours *run-time*. Considering that single `reduceAllTac` applications take up to 20 minutes to run, doing these proofs is extremely frustrating. The proof is only computationally demanding; no real intellectual effort is required. Proving these results on faster hardware is no real solution; an inherently more effective manner of proof is required. The hardware synthesis function `nadd` was not verified because the task would have taken a substantial amount of time. Conceptually, it is very simple; provide the appropriate functions to describe N bit vectors, map bit vectors to natural numbers, *etc.* Then perform an induction on the size of the adder. In practice this, coupled with the parametricity on the one bit adder, made the prospect too daunting. Moreover, no new interesting problems were expected to surface during the proof.

The embedded operational semantics rules turned out to be the most useful, which was unexpected. The rigid format to which the rules adhere make them very efficient to manipulate. Even if individual tactics such as `safeOpSemAllTac` take a substantial time to run, the result remains in a trusted format. The behaviour of the tactics and operations we provide is very predictable, which is essential in large proofs. The use of rewriting outside the embedded operational semantics has the opposite characteristics; generally the proofs take a long time, rewriting results in huge expressions which are hard to comprehend. Although these proofs roughly follow the same strategy[6], at a local level very particular lemmas often

[6] Often something along the line of: induction on time, obtain appropriate properties about well-typedness, compute the output of subexpressions, instantiate induction hypotheses, rewrite

have to be proved to coax the expression into just the right format.

It was a disappointment that the use of specifications for hierarchical simulation (Section 5.1.5) turned out to be harder than expected.

As future work, it is possible to consider some of the examples or applications in this chapter in more detail. We lacked the time to investigate a number of issues satisfactorily. It would be interesting to consider a medium sized case study (*e.g.* the Tamarack microprocessor [82].) However, considering the time very small examples took, we are pessimistic about a favourable outcome. We must, however, reiterate a point made earlier; picoELLA was designed to be a minimal language. To use it even in medium sized examples would be taking it out of the prototyping domain for which it was intended.

A fairly small effort would be required to incorporate the embedded operational semantics in the LAMBDA browser [64, Chapter 4]. The fixed set of rules and tactics make it an ideal candidate for a menu-based interface. This would be a very user-friendly introduction to the use of an operational semantics, with the proof system keeping track of side conditions *etc.*

`reduce` expressions, clean up.

Chapter 6

Conclusions and Future Work

First we summarise the work presented in this thesis. This is followed by some suggestions about possible future work. Finally we conclude.

6.1 Summary

In the introduction we gave a very brief overview of the evolution of hardware testing through the use of increasingly powerful hardware simulators. The combinatorial explosion of the number of test vectors to be simulated has forced alternative verification techniques to be developed. The field of formal hardware verification attempts to address this problem. A number of different methodologies are being used, but in this thesis we concentrated on the use of automated proof systems to formally verify hardware. However, the use of non-standard formal notations, while suited to hardware verification, has prevented industrial take-up of formal hardware verification techniques. One objective of this thesis is to investigate how this may be achieved.

In Chapter 2 we described how hardware description languages (HDLs) were initially used to document circuit designs. Following this, HDL descriptions were used to simulate circuit designs. The relation between the description of the hardware design and the behaviour of the implementation of this design was investigated.

The separation of the structure and the behaviour of a design is a crucial idea in this thesis. We indicated how this issue has also been studied in programming language semantics research. Two methods of relating structure and behaviour were presented in Sections 2.2 and 2.3. The former section shows how it is possible to extract a behaviour from a circuit description directly. Research taking this approach is reviewed from an evolutionary point of view. In Section 2.3 we present the methodology used in this thesis. Behaviour may be derived from a circuit description using a formal semantics; in particular an operational semantics. Automated proof systems may be used to provide automated and safe reasoning support in both extraction and derivation of behaviour.

As an initial step towards fulfilling the aim of this thesis, we combined the formal semantics of a simple hardware description language with a proof assistant. In doing so we obtained a versatile system which allows a number of paradigms to be integrated. These include: formal symbolic hardware simulation, proven transformations on circuits, interactive formal synthesis, verified hardware synthesis functions, and conventional hardware verification.

As a first step towards this goal, we defined a small HDL in Chapter 3. Two candidate HDLs were reviewed and compared, and a static and dynamic structural operational semantics of the picoELLA language was presented.

In Chapter 4 we introduced the proof system LAMBDA, used to automate the picoELLA semantics. Circuits were represented by abstract data type terms. The static and dynamic semantics were embedded as functions operating on circuits. The main theoretical result of this thesis was described in some detail in this chapter. It states that the embedded dynamic semantics of picoELLA is total and monotone in its arguments, and preserves a number of invariants. (Appendix B gave an overview of other theorems and lemmas.) We briefly discussed limitations of the LAMBDA proof system also.

A number of small examples were presented in Chapter 5. The embedded semantics of picoELLA was used to derive operational semantics rules closely resembling those presented in Chapter 3. In conjunction with meta-variables of the LAMBDA system these turned out to be an efficient (relative to other examples in this thesis) method to implement symbolic simulation. Formal interactive synthesis, both top-down and bottom-up, turned out be an extension of this approach. Hardware synthesis functions were described in Section 5.2.3. In Section 5.3 formally proved correct transformations on hardware descriptions were applied to a simple parity checker. Limitations of the usability of the embedding of picoELLA were discussed in Section 5.4. Expressions became very large quickly, which resulted in unacceptably long times to verify even small circuits. The use of a semantics to derive a behaviour from a structural description, as opposed to having a behavioural description, was also cumbersome.

Appendix A contains an overview of notation and terminology used in the thesis. An overview of theorems and lemmas is given in Appendix B. The embedded derived operational semantics rules, an alternative encoding of rules dealing with recursion, and a correspondence between the embedded rules and those of Sections 3.3.2 and 3.3.3 are described in Appendix C.

6.2 Future Work

An interesting outstanding issue is that of the 'maximal ELLA semantics,' presented in Section 3.3.5. There are some alternative semantics which are more defined than our semantics, and to prove possible relationships between them could be useful. The alternative semantics and their relationships may possibly be automated in LAMBDA.

Although the embedded operational semantics rules of Section 5.1 were the most efficient manner to simulate circuits, they do not seem to be so appropriate for proving properties by standard logical reasoning. More conventional correctness proofs, *e.g.* for the parity checker `PC`, were problematic. More work is needed to see if these proofs are fairly uniform and some proof support may be provided, or whether they are inherently inefficient. We hope that the former is the case, and tactics and derived rules can alleviate some of the problems.

It is probably not practical to use the interactive synthesis methods of Sections 5.2.1 and 5.2.2 for large examples. However, some work could be done to see where and how problems arise. Hardware synthesis functions such as `nadd` are probably much more useful. As explained in Section 5.5, we did not attempt to verify `nadd`, but this should be done, if only to see how the proof compares to other hardware description techniques.

Transformations on circuits are potentially very powerful, and more work must be done to see if transformations need to be as powerful as `fe` on page 220. It could well be that transformations not involving time are sufficient for most applications.

The examples of Chapter 5 were all very brief. A medium sized case study (*e.g.* the Tamarack microprocessor [82]) would be interesting to consider. Doing a larger example might show a useful uniformity in proofs.

The embedded operational semantics rules can be given a menu-based front-end by using the LAMBDA browser. This would make the operational semantics very easy to use.

6.3 Conclusions

We hope to have given fresh insights into the relationship between the description of hardware designs and the corresponding behaviour. Different approaches were presented in Chapter 2 culminating in a totally formal approach to deriving behaviours from circuit descriptions using an operational semantics.

The work presented in this thesis is new in aiming to integrate an existing hardware description language and a formal proof system. Until recently little research was carried out which intended to combine a hardware description notation and a proof system by using an embedded formal semantics. The semantics for a subset of ELLA formalised a previously existing model. This subset picoELLA is the only HDL embedded in a proof system we are aware of to allow undelayed feedback loops. The embedding of this semantics in LAMBDA raised a number of points of interest about bottom values in the semantics. Proofs of meta-theoretic properties about the embedded semantics, rather than properties about particular objects are significant. The main theorem of this thesis, described in Section 4.3.2, is that the semantic model computes the least fixed point solution of the circuit. By showing that the shape of circuits remains unchanged over time, we can conclude that only the state of delays needs to be stored to characterise the circuit as it evolves over time. The relationship between the standard dynamic semantics and the greatest lower bound semantics (Sections 3.3.5 and 4.3.3) has also been proved formally.

A number of small examples were presented to assess the claim to be able to use a number of different paradigms within a proof system. Although this seems to be the case in theory, practical problems prevented even medium sized examples being carried out. Proving meta-results about the embedded semantics presented no problems, but reasoning about individual circuits turned out to be hard. The particular form of semantics was partly responsible; including the state of a circuit in its description made theorems messy to state. However, no comparison was made with the alternative (using a mutable state, and fixed circuit) so that any judgement cannot be final. Another problem was that terms became too large very quickly, which made intermediate expressions and rules unwieldy.

By providing a theoretical basis for combining a hardware description language and a proof system, we hope that this work has been useful as a first step towards providing formal tools which can be used successfully outside of academia.

Appendix A

Glossary of Terminology and Notation

A.1 Terminology

Logical frameworks are systems in which particular logics may be specified. These logics may then be reasoned in or about using the logical framework. The Edinburgh LF [89], implemented in LEGO [116] is an example of such a system.

Proof systems are implementations of a particular logic, such as higher-order logic.

Rewrite systems are programs implementing the automatic application of a particular set of rewrite rules. An example is OBJ [68].

Theorem provers are proof systems which are capable of performing substantial proofs without user intervention such as the Boyer-Moore system [21], OTTER [34], CLIO [12] and the Larch Prover [166].

Proof assistants are proof systems which may perform small proofs with no user intervention but are normally used in an interactive mode, driven by the user. These include HOL [79], VERITAS [88,87], LAMBDA [64], NUPRL [46,102].

Proof checkers are proof assistants which have to be supplied with all details of a proof. Their function is to validate proofs. An example is the Stanford LCF system [127].

A.2 Notation

This section describes the notation adopted in the thesis for displaying LAMBDA rules and LAMBDA output.

We display LAMBDA constructs in this thesis according to the following table. Where no explicit page is mentioned consult the LAMBDA reference manual [64, Section 2.1.1.1].

LAMBDA *Output*

<table>
<tr><th>Displayed As</th><th>LAMBDA Syntax</th><th>Meaning</th><th>Page</th></tr>
<tr><td>$\vdash$</td><td><code>|-</code></td><td>entailment</td><td></td></tr>
<tr><td><code>P#(</code>x<code>)</code></td><td><code>P#(x)</code></td><td>abbreviation or context</td><td>87</td></tr>
<tr><td>ιx. <code>P#(</code>x<code>)</code></td><td><code>iota x. P#(x)</code></td><td>Iota operator</td><td>89</td></tr>
<tr><td>$\forall x$. <code>P#(</code>x<code>)</code></td><td><code>forall x. P#(x)</code></td><td>universal quantification</td><td></td></tr>
<tr><td>$\exists x$. <code>P#(</code>x<code>)</code></td><td><code>exists x. P#(x)</code></td><td>existential quantification</td><td></td></tr>
<tr><td>$\rightarrow$</td><td><code>->></code></td><td>implication</td><td></td></tr>
<tr><td>$\leftrightarrow$</td><td><code><-></code></td><td>biimplication</td><td></td></tr>
<tr><td><code>E</code></td><td><code>E</code></td><td>existence predicate</td><td>88</td></tr>
<tr><td>$\wedge$</td><td><code>/\</code></td><td>truth-valued conjunction</td><td></td></tr>
<tr><td>$\vee$</td><td><code>\/</code></td><td>truth-valued disjunction</td><td></td></tr>
<tr><td><code>NOT</code></td><td><code>NOT</code></td><td>truth-valued negation</td><td></td></tr>
<tr><td><code>fn</code> x <code>=> P#(</code>x<code>)</code></td><td><code>fn x => P#(x)</code></td><td>object-level lambda expression</td><td>87</td></tr>
<tr><td><code>lam</code> x. <code>P#(</code>x<code>)</code></td><td><code>lam x. P#(x)</code></td><td>meta-level lambda expression</td><td>87</td></tr>
<tr><td><code>===</code></td><td><code>===</code></td><td>equivalence</td><td>88</td></tr>
<tr><td><code>==</code></td><td><code>==</code></td><td>truth-valued equality</td><td>88</td></tr>
<tr><td><code>=</code></td><td><code>=</code></td><td>boolean equality</td><td>89</td></tr>
</table>

The operators are ordered in decreasing precedence as follows: `=`, `NOT`, `E`, `==`, `===`, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\vdash$.

As an example, the following rule with two premises, each with some hypotheses

```
******* LEVEL 2 *******
2:  E x $ E y $ G // R#(x,x) $ H |- P \/ R#(x,NOT y)
1:  E x $ E y $ G // H |- P /\ Q
    --------------------
    E x $ E y $ G // P /\ Q ->> R#(x,x) $ H |- P \/ R#(x,NOT y)
```

will be displayed as

```
***** Level 2 *****


***** Premise 2 *****
1:  E x
2:  E y
1:  R#(x,x)
⊢ P ∨ R#(x,NOT y)
***** Premise 1 *****
1:  E x
2:  E y
⊢ P ∧ Q
--------------------
1:  E x
2:  E y
1:  P ∧ Q → R#(x,x)
⊢ P ∨ R#(x,NOT y)
```

Note that the G and H lists are numbered separately, with the G list displayed first. The trailing `G` and `H` are omitted. Sometimes we will omit the numbering of the premises and hypotheses. When listing theorems we will often omit the dashed line. Expressions of the form `B == true` will often be displayed as `B`, but `B == false` will always be displayed in the original form. Existence hypotheses may be left out, but this will be mentioned explicitly. The `typewriter` font is used for function definitions, data type constructors, ML keywords, and meta-variables representing meta-level objects (*e.g.* abbreviations.) *Italic* font is used for meta-variables representing object-level objects, and variables in patterns. There are some additional pretty-printing conventions which are used for embedded operational semantics rules; these are described in Appendix C.

A.3 Overview of Types and Functions Used in this Thesis

*pico*ELLA *Semantics Type Definitions*

Type	Typical Element	Page
Env	Γ	67
$Type$	τ	67
$TEnv$	T	67
$AEnv$	S	67
$\mathbb{N}$	i, j, n	

The typical elements may also be primed and subscripted. For the BNF definition of picoELLA on page 66, the typical element of a syntactic class has the same name as the syntactic class, possibly subscripted or primed. In addition, c is also used as an element of the syntactic class *const*.

*pico*ELLA *Semantics Definitions*

Symbol	Description	Page
$\downarrow$	projection of constants to bottom	68
ff	Kleene's falsity value	68
$match$	matching function	68
tt	Kleene's truth value	68
uu	Kleene's undefined value	68

Types Used in The Embedding

LAMBDA Type	Description	Page
'a list	lists	[64, Section 3.17]
biop	built-in operators	229
bool	boolean values	[64, Section 3.12],102
bool3	Kleene's ternary logic	123
chooser	picoELLA choosers	113
const	picoELLA constants	105
expr	picoELLA expressions	115
natural	natural numbers	[64, Section 3.13]
tpe	picoELLA types	106
tree	binary tree (example)	90

Typical Elements Used in The Embedding

LAMBDA Type	Typical Element
'a list	*l*, *m*
const list	*l*, *m*, *env*, *instream*, *outstream*
tpe list	*env*
bool	*b*
chooser	*ch*, *chooser*
const	*c*, *d*, *i*, *o*, *v*, *out*, *initial*
expr	*e*, *f*, *g*, *circ*, *newcirc*
natural	*n*, *m*, *t*
tpe	*s*, *t*

env may be used with the letter *l* or *r* appended. *c* and *e* may be used in the form *cil* and *cir* where *i* is a natural number. All elements may be used primed, with an underscore and/or natural number appended, *e.g.* *o_*, *o1_1*, *circ'_*, *circ'_1'*. A trailing prime indicates that the variable is restricted [64, Section 2.19.3]. The type of a variable (*e.g.* for *l* or *t*) will always be clear from the context.

Functions Used in The Embedding

LAMBDA Name	Description	Page
`&&`	boolean conjunction	[64, Section 3.12]
`||`	boolean disjunction	[64, Section 3.12]
`abs`	boolean to constant representation	189
`absinv`	constant to natural number abstraction	189
`absInv`	inverse of `abs`	200
`again`	iterative part of dynamic semantics of expressions	125
`againGlb`	iterative part of alternative dynamic semantics of expressions	163
`and3`	conjunction of Kleene's ternary logic	123
`bottomOfConst`	constant to bottom function	109
`ceq`	boolean equality on constants	107
`cheq`	boolean equality on choosers	114
`cle`	data ordering on constants	108
`elem`	element lookup in list	123
`enumToNat`	constant to natural coercion	228
`eq`	boolean equality on naturals	107
`eq3`	Kleene's equality, gives `tt` and `ff`	123
`even`	even-ness predicate	189
`fe`	initial delay value transformation on circuits	220
`fe'`	initial delay value transformation on circuits	222
`ff`	Kleene's falsity value	123
`glb`	greatest lower bound	108
`iterate`	iterative part of dynamic semantics of expressions	124
`iterateGlb`	iterative part of alternative dynamic semantics of expressions	163
`leaves`	number of leaves of a tree (example)	93
`le3`	data ordering in Kleene's logic	123

Functions Used in The Embedding (Cont'd)

LAMBDA Name	Description	Page
`lle`	data ordering on lists, extended from `cle`	108
`match`	matching algorithm	123
`nadd`	N bit adder synthesis function	211
`nodes`	number of nodes of a tree (example)	93
`noof`	count number of vs in input	189
`not`	boolean negation	102
`not3`	negation of Kleene's ternary logic	123
`or3`	disjunction of Kleene's ternary logic	123
`ple`	data ordering on expressions	122
`plus`	addition of picoELLA integers	229
`reduce`	dynamic semantics of expressions	124
`reduceGlb`	alternative dynamic semantics of expressions	163
`reduceSeq`	dynamic semantics of expressions	126
`replicate`	removal of fan-out in circuits	223
`right`	function definition example	93
`shapeEq`	equal shape predicate on expressions	121
`sizeOfConst`	size function on constants	110
`sizeOfExpr`	size function on expressions	118
`state`	state of parity checker	189
`sub`	function definition example	87
`suspend`	part of dynamic semantics of expressions	195
`tt`	Kleene's truth value	123
`typeEq`	boolean equality on types	108
`typeOfChooser`	static semantics of choosers	113
`typeOfConst`	static semantics of constants	106
`typeOfExpr`	static semantics of expressions	119
`typeOfExprIter`	alternative static semantics of expressions	248
`uu`	Kleene's undefined value	123
`wrong`	function definition example	93

Abbreviations Used in The Embedding

LAMBDA Name	Description	Page
`ADD_SPEC`	half adder specification	200
`ADD`	picoELLA full adder	199
`AND`	picoELLA AND gate	179
`BEHEQ`	behavioural equivalence for transformations	215
`BEQ`	correctness statement for transformations	215
`bit`	the undefined value of type bit	106
`CORRECT`	correctness statement for transformations	219
`BITINDUCT`	definition of case analysis on bits	111
`DEFBITINDUCT`	definition of case analysis on defined bits	111
`FF`	picoELLA flip-flop	193
`FIXPOINT`	definition of fixed point	162
`HA`	picoELLA half adder	199
`HA_SPEC`	half adder specification	200
`hi`	a value of type bit	106
`HOLPC`	picoELLA definition of a HOL parity checker	186
`LEASTFIXPOINT`	definition of least fixed point	162
`lo`	a value of type bit	106
`MUX`	picoELLA multiplexor	186
`NOT_g`	picoELLA NOT gate	186
`OR`	picoELLA OR gate	199
`PC`	picoELLA parity checker	188
`REG`	picoELLA register	186
`THM`	`reduce`'s monotonicity theorem	148
`THMI`	iterative part of monotonicity theorem	147
`THMR`	reduction part of monotonicity theorem	137
`WF`	well-formedness predicate on transformations	215
`XOR`	picoELLA XOR gate	199

Appendix B

Overview of Lemmas and Statistics

B.1 Overview of Lemmas Proved

In Section 4.1.2 we described how the statement of an interesting fact as a theorem is often not very useful. Thus, corresponding to a theorem 'T' statement, there is an 'R' version which has all quantifiers stripped off, and implications and conjunctions moved to the hypothesis. (Our format for 'R' and 'L' rules is different from those used in the LAMBDA documentation [64, Section 3.1].) An 'L' version introduces the conclusion on the left hand side. For certain results we also have a 'U', 'F', and 'E' version. The unfold 'U' version is used to replace subexpressions in the conclusion, and the fold 'F' version the inverse rule. The 'E' version is an equality or equivalence rule which can be used in rewriting. We illustrate this naming convention with `bottomOfConstPreservesTypeT`:

```
***** bottomOfConstPreservesTypeT *****
--------------------
⊢ ∀c.  typeEq (typeOfConst c) (typeOfConst (bottomOfConst c))
```

```
***** bottomOfConstPreservesTypeR *****
--------------------
E c ⊢ typeEq (typeOfConst c) (typeOfConst (bottomOfConst c))


***** bottomOfConstPreservesTypeL *****
1:  E c
1:  typeEq (typeOfConst c) (typeOfConst (bottomOfConst c))
⊢ P
--------------------
E c ⊢ P


***** bottomOfConstPreservesTypeU *****
E c ⊢ P#(typeOfConst c)
--------------------
E c ⊢ P#(typeOfConst (bottomOfConst c))


***** bottomOfConstPreservesTypeE *****
--------------------
⊢ typeOfConst (bottomOfConst c) === when#(E c,typeOfConst c)
```

In total, more than 700 results have been proved, but most of the 'T' versions can be summarised in the table below.

A function is total (**T**), or total only if its arguments are well-typed (**W**.) **C**, **T**, **R**, **A**, and **I** stand for commutativity, transitivity, reflexivity, associativity, and idempotency of the function in the appropriate columns. An entry $*$ (or a particular number n) for monotonicity shows that the function is monotone for all arguments (or the nth argument, respectively.) In the column for invariants **C**, **E**, and **S** mean that the function preserves the type of constant inputs, the type of expression inputs, and shape of expression inputs. **E** and **S** preserve type (shape) both between two inputs, and across the function call, *cf.* `reduce`'s monotonicity statement on page 137. `=` indicates that if the function is reflexive for two arguments, the arguments are equal. `==` indicates that the function is an encoding of the equality `==`.

Function	Total	Comm	Trans	Refl	Mono	Invar	Other
`not3`	**T**				$*$		1
`or3`	**T**	**C**			$*$		**A**
`and3`	**T**	**C**			$*$		**A**
`eq3`	**T**	**C**	**T**	**R**			`==`
`ceq`	**T**	**C**	**T**	**R**			`==`
`cheq`	**T**	**C**	**T**	**R**			`==`
`typeEq`	**T**	**C**	**T**	**R**			`==`
`shapeEq`	**T**	**C**	**T**	**R**			2
`le3`	**T**		**T**	**R**			`=`
`cle`	**W**		**T**	**R**			`=`,3
`ple`	**W**		**T**	**R**			`=`,4
`lle`	**T**		**T**	**R**			
`glb`	**W**	**C**			$*$	**C**	**I**,5
`bottomOfConst`	**T**					**C**	**I**,6
`typeOfConst`	**T**						
`typeOfChooser`	**T**						
`typeOfExpr`	**T**						
`sizeOfConst`	**T**						3
`sizeOfExpr`	**T**						4
`match`	**W**				2		7
`reduce`	**W**				$*$	**C**,**E**,**S**	8
`reduceSeq`	**W**				$*$	**C**,**E**,**S**	
`iterate`	**W**				$*$	**C**,**E**,**S**	
`reduceGlb`	**W**				$*$	**C**,**E**,**S**	9
`iterateGlb`	**W**				$*$	**C**,**E**,**S**	

1 de Morgan's rules and a number of rewrite rules to provide a coherent rewrite strategy for the three valued boolean operators have also been proved.

2 The relation between `shapeEq` and `typeOfExpr` is very uncomfortable. If two expressions are both well-typed and shape equal, their types are the same. But a well-typed expression which is shape equal to another expression does not imply that the other expression is also well-typed. The reason is that initial approximations in `LetRec` constructs must be equal to bottom to be well-typed, but `shapeEq` only ensures that type types are the same (see pages 119 and 121.) We can define a more liberal version of `typeOfExpr` which is preserved by `shapeEq`. `typeOfExprIter` differs from `typeOfExpr` only in the `LetRec` clause:

```
fun typeOfExprIter te (LetRec (c, e1, e2)) =
        (fn tc =>
        (fn (t1, b1) =>
        (fn (t2, b2) =>
                (t2, b1 && b2 && (typeEq t1 tc)))
        (typeOfExprIter (tc::te) e2))
        (typeOfExprIter (tc::te) e1))
        (typeOfConst c) | ...
```

Obviously well-typedness using `typeOfExpr` implies `typeOfExprIter`, but not *vice versa*. We can now prove that

$\vdash \forall e,f,l,t.$ `shapeEq` e f $\wedge$ `typeOfExprIter` l e `==` $(t,$`true`$)$ $\rightarrow$
`typeOfExprIter` l f `==` $(t,$`true`$)$

If we also encode a `bottomOfExpr` function, which projects the initial values of recursive `LET`s to bottom, we obtain a precise statement:

$\vdash \forall l,e.$ `typeOfExpr` l (`bottomOfExpr` e) `==` `typeOfExprIter` l e

3 A useful theorem relating `cle` and the natural number ordering `<=` on the corresponding sizes of the arguments is the following:

$\vdash \forall c, d.$ `typeOfConst` c `==` `typeOfConst` $d \rightarrow$
`cle` c $d \rightarrow$ `sizeOfConst` d `<=` `sizeOfConst` c

This is useful to know when proving the totality of the `iterate` and `reduce` functions, for example. See also the discussion about the relation of the data ordering and the size of constants in Section 4.2.1.

4 `ple` is total on arguments which have the same shape only.

$\vdash \forall e, f.$ `shapeEq` e $f \rightarrow$ `E (ple` e f`)`

As `cle` above, `ple` e f implies `>=` on the corresponding sizes of e and f.

5 `glb` is less than both of its arguments.

6 A bottom value is less than any other value of the same type, so that being less than bottom implies being bottom. For every type a corresponding bottom value exists.

7 If we abbreviate $\forall c.$ `match` ch c `==` `match` ch' c by match equality `meq` ch ch' we have the following results.

$\vdash \forall x, y.$ `meq (`x `|` x`)` x
$\vdash \forall x, y.$ `meq (`x `|` y`) (`y `|` x`)`
$\vdash \forall x, y, z.$ `meq ((`x `|` y`) |` z`) (`x `| (`y `|` z`))`
$\vdash \forall x, y, z.$ `meq (`x `|` y`,` z`) ((`x`,` z`) | (`y`,` z`))`
$\vdash \forall x, y, z.$ `meq (`x`,` y `|` z`) ((`x`,` y`) | (`x`,` z`))`

The monotonicity of `match` and a corollary stating the inability of match to distinguish between `cle`-related constants are described in Section 4.3.1.

8 In Section 4.3.3 we showed a number of corollaries of the `reduce` monotonicity theorem. We mentioned, but did not show the following result.

$\vdash \forall envl\_, c0l\_, c0r\_, e0l\_, c1l\_, c1r\_, e1l\_, e1r\_, c2l\_, e2l\_.$

```
typeOfExpr (map typeOfConst (c0l_::envl_)) e0l_ ==
                 (typeOfConst c0l_,true) ∧
reduce (c0l_::envl_) e0l_ == (c1l_,e1l_) ∧
cle c0l_ c1l_ == true ∧
iterate envl_ e0l_ c0l_ == (c2l_,e2l_) ∧
reduce (c0r_::envl_) e0l_ == (c1r_,e1r_) ∧
cle c0r_ c1r_ == true ∧
typeOfConst c0l_ == typeOfConst c0r_ ∧
cle c0l_ c0r_ == true ∧
cle c0r_ c2l_ == true → iterate envl_ e0l_ c0r_ == (c2l_,e2l_)
```

The theorem `cleImpliesSameFixpointT` states that given initial approximation *c0l_*, second approximation *c1l_* and their fixed point *c2l_*, if there is another value *c0r_* with next approximation *c1r_*, such that *c0r* lies between the first value *c0l* and the fixed point *c2l*, then *c0r_* evaluated to the same fixed point. It is unfortunate that we need to mention *c1r_*, but we need to ensure that *c0r_*'s fixed point computation terminates.

9 In Section 4.3.3 we showed the theorem stating that `reduceGlb` is a more optimistic semantics than `reduce`. A similar theorem has been proved relating `iterate` and `iterateGlb`. All results of `reduce` and `iterate`, such as those involving fixed points, can also be proved for `reduceGlb` and `iterateGlb`.

B.2 Some Statistics about Proofs

The proof scripts, containing more than 14,000 lines with over 10,000 rule applications and 2,800 tactic applications, take nearly 13 hours to run on an unloaded sun 4/65 (sparc station 1+) with 32M memory and a local hard disk. A maximum size (48M) heap was used. LAMBDA does not provide any facilities to time proofs, count inferences, rules, *etc.* We provided an alternative startup database where functions such as `apprl`, `applyTac` have been redefined to only count the inferences carried out, and ignore the computation of the proof. For example, `doRules` is redefined to add the number of rules it normally applies to the rule counter. Unfortunately, it is not possible to count basic inferences, or the number of rewrite rules which would be applied during the course of a proof. Nevertheless we obtain some interesting results. This is the output from the counting functions after running all proof scripts:

```
10539          rule applications have been counted.
5047           rewrite rules have been counted (upon declaration.)
17893          rewrite rules have been explicitly used in proofs.
2814           tactic applications have been counted.
4699           permutation applications have been counted.
733            popGoals have been counted.


3.84           tactics per proof.
14.38          rules per proof.
24.41          rewrite rules explicitly supplied per proof.
6.411          permutations per proof.
3.75           rules per tactic.
6.36           rewrite rules per tactic.
1.67           permutations per tactic.
5.42           permutations + rules per tactic.
0.45           permutations:rule ratio.
1.48           G permutations:H permutations ratio.
```

The number of rule applications excludes rewrite rules, and counts derived rule applications only. There is no way of counting the number of basic inference rules which have been used. The explicitly used rewrite rules arise from passing rewrite rules into rewrite tactics. This figure is a substantial underestimate because derived tactics, such as `typeOfExprTac`, are not included in this count. A total of

5047 rewrite rules were used in the declarations of such tactics. `typeOfExprTac`, which uses 13 specialised rules in addition to the standard rewriting rules, was used 122 times in the proof scripts. This loses 1586 rewrite rules in the count above. For a larger tactic `reduceAllTac'` this figure is $284 \times 38 = 10792$. Of course, we still cannot say how many of these rules are actually used in a proof, nor how many times.

We were surprised to see how high the permutation per rule ratio was. In effect this means that for roughly every two rules we had to move something in the hypothesis lists. This is an enormous overhead. Moreover, the ratio of G list (for existence conditions) to H list (for 'conventional' hypotheses) permutations was 1.5. It seems that in a logic which does not have partial terms the proofs would have contained 60% percentage less permutation commands. The lemmas proving totality of all functions which are essential in LAMBDA version 3.2, would also disappear.

If we consider the fact that for every theorem there are at least two other very simple rules proved (the 'R' and 'L' versions) the figures become somewhat more realistic. We multiply all 'per proof' figures times three, giving

10.17	tactics per proof.
43.13	rules per proof.
73.23	rewrite rules explicitly supplied per proof.
19.23	permutations per proof.

Another problem is the variation in the sizes of proofs. A lot of simple proofs consisted of one tactic each, but the most complicated proof took more than 300. Averaging out these values does not give a true reflection of the amount of effort which went into more interesting, larger, proofs. The figures from the monotonicity of `reduceGlb` alone are as follows.

1437	rule applications have been counted.
2304	rewrite rules have been explicitly used in proofs.
320	tactic applications have been counted.
735	permutation applications have been counted.
4.49	rules per tactic.
7.2	rewrite rules per tactic.
2.30	permutations per tactic.
6.79	permutations + rules per tactic.
0.51	permutations:rule ratio.
2.09	G permutations:H permutations ratio.

The first four numbers can be set against the first set of average 'per proof' numbers. The last six figures can be compared with those computed for all the proof scripts because they are stated per tactic or rule, which is independent of the size of the proof. It is interesting to note that the number of rules, rewrite rules and permutations per tactic is higher than the average. This means that large proofs are not only longer, they are also more complicated on a per tactic basis.

All the figures above exclude examples from the case studies of Chapter 5. Most of the examples are not very long, but tend to be very CPU intensive. For example, the four bit adder example of the embedded operational semantics rules on page 213 has the following proof for the derived rule.

```
flexn 3;
flex' (grn 2(gL([gI],gI)));
apprl reduceNADDSSn;
atn [2] (doRule reduceNADDSSn);
atn [4] (pureRewriteTac [betaRedE,exLemma1,whenTrueE]);
atn [11] (pureRewriteTac [betaRedE,exLemma2,whenTrueE]);
ata (safeOpSemAllTac'[reduceNADD2bit'',reduceADD1]);
atn [2,3,5,6,8,9] (doRules [reduceCoTuple,reduceCons,reduceType,
                             reduceSn,reduce0]);
atn [2,3,4,5] safeOpSemAllTac;
```

It takes just under 20 minutes to run due to the enormous amount of work `safeOpSemAllTac` performs. This pales into insignificance if we try to prove that the full adder `ADD` of page 200 satisfies its specification using standard rewriting. We would have to consider 8 inputs, which each take more than 40 minutes to rewrite.

Appendix C

Embedded Operational Semantics Rules

In this appendix we define the pretty printing conventions for the embedded operational semantics rules, which are then listed. Some alternative embedded operational semantics rules for dealing with recursion are then presented. Following this, a correspondence between paper and embedded operational semantics rules is given. Finally some remarks are made about the difference between the forward and backward rule application of the semantic rules, and why this leads to some changes to the rules used in a forward manner.

C.1 Pretty Printing Conventions

The embedded operational semantics rules are pretty printed as all other LAMBDA output with the following additional transformations. All expressions of type *expr* have been manually converted from a prefix notation, *e.g.* `Let` (e,f), to an infix notation `LET` e `IN` f. To code this in LAMBDA as part of the pretty printing functions would require a considerable effort. Moreover, expressions involving `typeOfConst`, `typeOfChooser`, `typeOfExpr`, `reduce`, `iterate`, and `reduceSeq` are printed in a more conventional operational semantics syntax. The resultant output reflects accurately the operational semantics definition outside the proof system (see Section 3.3.)

$(instream\_,\ env\_ \vdash circ\_ \Rightarrow (outstream\_, circ\_))$

is an abbreviation for

`reduceSeq` $env\_\ circ\_\ instream\_$ `==` $(outstream\_, circ\_)$

And

$(env\_ \vdash circ\_ \Rightarrow (o\_, circ'\_) :\ t\_)$

is an abbreviation for

`typeOfExpr (map typeOfConst` $env\_$`) == (`$t\_$`,true)` $\wedge$
`reduce` $env\_\ circ\_$ `==` $(o\_, circ'\_)$

And

$(o\_,\ env\_ \vdash circ\_ \Rightarrow (o1\_, circ'\_) :\ t\_)$

is an abbreviation for

`typeOfExpr (map typeOfConst` $env\_$`) == (`$t\_$`,true)` $\wedge$
`iterate` $env\_\ circ\_\ o\_$ `==` $(o1\_, circ'\_)$

Moreover, expressions involving `typeOfConst` have been converted to the postfix notation $x : t$. `typeOfChooser` and `typeOfExpr` expressions have been converted to a postfix $x : (t, b)$ notation.

The following abbreviation is used for the alternative formulation of the recursion rules of Section C.3.

$(o\_,\ o1\_,\ env\_ \vdash circ\_ \Rightarrow (o2\_, circ2\_) :\ t\_)$

is an abbreviation for

`typeOfExpr (map typeOfConst` $env\_$`) == (`$t\_$`,true)` $\wedge$
`suspend` $env\_\ circ\_\ o\_\ (o1\_, circ1\_)$ `==` $(o2\_, circ2\_)$

C.2 The Embedded Operational Semantics Rules

`reduceSeqNil` terminates the simulation, when there are no more input values to be processed.

```
***** reduceSeqNil *****
--------------------
⊢ ([], env_ ⊢ circ_ ⇒ ([],circ_))
```

The `reduceSeqCons` rule advances time, and takes the first value of the input stream and pushes it onto the environment.

```
***** reduceSeqCons *****
⊢ (instream_, env_ ⊢ circ1_ ⇒ (outstream_,circ2_))
⊢ (i1_ :: env_ ⊢ circ_ ⇒ (o1_,circ1_): t_)
--------------------
⊢ (i1_ :: instream_, env_ ⊢ circ_ ⇒ (o1_ :: outstream_,circ2_))
```

The following rule starts the computation of the fixed point of the `LET REC`. The third premise states that the initial approximation *initial_* must be equal to the bottom value (`bottomOfConst` *initial_*). The type *t1_* of the initial approximation must be equal to the type of the defining expression.

```
***** reduceLetRec *****
⊢ E t1_
⊢ o1_: t1_
⊢ initial_: t1_
⊢ bottomOfConst initial_ == initial_
⊢ (o1_ :: env_ ⊢ circ2_ ⇒ (o2_,circ2'_): t2_)
⊢ (initial_, env_ ⊢ circ1_ ⇒ (o1_,circ1'_): t1_)
--------------------
⊢ (env_ ⊢ LET INIT initial_ REC circ1_ IN circ2_ ⇒
(o2_,LET INIT initial_ REC circ1_' IN circ2_'): t2_)
```

The `reduceFix` rule detects a fixed point *initial_*.

```
***** reduceFix *****
⊢ (initial_ :: env_ ⊢ circ1_ ⇒ (initial_,circ1'_): t1_)
--------------------
⊢ (initial_, env_ ⊢ circ1_ ⇒ (initial_,circ1'_): t1_)
```

The third premise of `reduceIterate` determines that we have not yet reached a fixed point. It therefore iterates again in premise two, this time with the new approximation *o1_*. Recall that `ceq` is an encoding of equality on constants.

```
***** reduceIterate *****
⊢ ceq initial_ o1_ == false
⊢ (o1_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_): t1_)
⊢ (initial_ :: env_ ⊢ circ1_ ⇒ (o1_,circ1'_): t1_)
--------------------
⊢ (initial_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_): t1_)
```

The rule for the non-recursive `LET` is much simpler; we can just push the result of the defining expression onto the stack.

```
***** reduceLet *****
⊢ E t1_
⊢ o1_: t1_
⊢ (o1_ :: env_ ⊢ circ2_ ⇒ (o2_,circ2'_): t2_)
⊢ (env_ ⊢ circ1_ ⇒ (o1_,circ1'_): t1_)
--------------------
⊢ (env_ ⊢ LET circ1_ IN circ2_ ⇒ (o2_,LET circ1'_ IN circ2'_): t2_)
```

The following two rules deal with the lookup of variables, as encoded by the de Bruijn encoding.

```
***** reduceVarSn *****
⊢ (env_ ⊢ Var n ⇒ (o2_,Var n): t2_)
--------------------
⊢ (o1_ :: env_ ⊢ Var (S n) ⇒ (o2_,Var (S n)): t2_)
```

```
***** reduceVar0 *****
⊢ out_:  t_
--------------------
⊢ (out_ ::  env_ ⊢ Var 0 ⇒ (out_,Var 0):  t_)
```

The output from a delay is its state; its new state is the output from the expression *circ_*. The type of the state must be same as the type of the input expression.

```
***** reduceDelay *****
⊢ initial_:  t_
⊢ (env_ ⊢ circ_ ⇒ (out_,circ'_):  t_)
--------------------
⊢ (env_ ⊢ DELAY (initial_,circ_) ⇒
(initial_,DELAY (out_,circ'_)):  t_)
```

```
***** reduceTuple *****
⊢ (env_ ⊢ circ2_ ⇒ (o2_,circ2'_):  t2_)
⊢ (env_ ⊢ circ1_ ⇒ (o1_,circ1'_):  t1_)
--------------------
⊢ (env_ ⊢ (circ1_,circ2_) ⇒
(CoTuple (o1_,o2_),(circ1'_,circ2'_)):  TyTuple (t1_,t2_))
```

The derivation of the semantic rules for the `IF` statement use the fact that the semantics is total. At the time this work was carried out (July 1991) this result had not been proved for the embedding which included the `LET REC`. As a result the rules in this embedding were more complicated than the rules in the embedding without the `LET REC`, for which the totality result had been proved. The totality result discharges the subgoal $\vdash$ *out_* : *t_* in `reduceIf` and `reduceIf'` below. In a similar manner we should be able to discharge premises 5 and 4 of `reduceLetRec` and `reduceLet` respectively.

***** reduceIf *****

$\vdash$ E $t\_$

$\vdash$ $out\_$: $t\_$

$\vdash$ $chooser\_$: $t\_$

$\vdash$ ($env\_$ $\vdash$ $branch2\_$ $\Rightarrow$ ($o2\_$, $branch2'\_$) : $t1\_$)

$\vdash$ ($env\_$ $\vdash$ $branch1\_$ $\Rightarrow$ ($o1\_$, $branch1'\_$) : $t1\_$)

$\vdash$ ($env\_$ $\vdash$ $circ\_$ $\Rightarrow$ ($out\_$, $circ'\_$) : $t\_$)

$\vdash$ ($env\_$ $\vdash$ IF $circ\_$ MATCHES $chooser\_$ THEN $branch1\_$ ELSE $branch2\_$ $\Rightarrow$

(case match $chooser\_$ $out\_$ of

uu $\Rightarrow$ bottomOfConst $o1\_$ | tt $\Rightarrow$ $o1\_$ | ff $\Rightarrow$ $o2\_$,

IF $circ\_'$ MATCHES $chooser\_$ THEN $branch1\_'$ ELSE $branch2\_'$) : $t1\_$)

Note in `reduceIf` that the types of the two branches must be equal, and that the type of the chooser must match that of the selecting expression. `reduceIf` returns a symbolic answer, but most of the time we want a concrete value. `reduceIf'` therefore explicitly computes the output value $o3\_$.

***** reduceIf' *****

$\vdash$ E $t\_$

$\vdash$ $out\_$: $t\_$

$\vdash$ o3_ == (case match $chooser\_$ $out\_$ of

uu $\Rightarrow$ bottomOfConst $o1\_$ | tt $\Rightarrow$ $o1\_$ | ff $\Rightarrow$ $o2\_$)

$\vdash$ $chooser\_$: $t\_$

$\vdash$ ($env\_$ $\vdash$ $branch2\_$ $\Rightarrow$ ($o2\_$, $branch2'\_$) : $t1\_$)

$\vdash$ ($env\_$ $\vdash$ $branch1\_$ $\Rightarrow$ ($o1\_$, $branch1'\_$) : $t1\_$)

$\vdash$ ($env\_$ $\vdash$ $circ\_$ $\Rightarrow$ ($out\_$, $circ'\_$) : $t\_$)

$\vdash$ ($env\_$ $\vdash$ IF $circ\_$ MATCHES $chooser\_$ THEN $branch1\_$ ELSE $branch2\_$ $\Rightarrow$

(o3_,IF $circ\_'$ MATCHES $chooser\_$ THEN $branch1\_'$ ELSE $branch2\_'$) : $t1\_$)

`reduceIfTt`, `reduceIfFf`, `reduceIfUu` have not been listed. They correspond to the three possible outputs the `IF` statement can deliver, and each include a premise to show that it is the `THEN`, `ELSE` or undefined branch which is taken. There is a rule for each of the indexing operators.

```
***** reduceIndex1 *****
⊢ E t2_
⊢ (env_ ⊢ circ_ ⇒ (CoTuple (o1_,o2_),circ'_):  TyTuple (t1_,t2_))
--------------------
⊢ (env_ ⊢ circ_[1] ⇒ (o1_,circ'_[1]):  t1_)
```

```
***** reduceIndex2 *****
⊢ E t1_
⊢ (env_ ⊢ circ_ ⇒ (CoTuple (o1_,o2_),circ'_):  TyTuple (t1_,t2_))
--------------------
⊢ (env_ ⊢ circ_[2] ⇒ (o2_,circ'_[2]):  t2_)
```

The remainder of the rules deal with the static semantics proof obligations, which may arise from the previous rules.

```
***** reduceCoTuple *****
⊢ d:  t2_
⊢ c:  t1_
--------------------
⊢ CoTuple (c,d):  TyTuple (t1_,t2_)
```

```
***** reduceCons *****
--------------------
⊢ Cons (n,m):  Type m
```

The following three rules deal with typing of choosers.

```
***** reduceT *****
⊢ ch2_:  t2_
⊢ ch1_:  t1_
--------------------
⊢ T (ch1_,ch2_):  TyTuple (t1_,t2_)
```

```
***** reduceB *****
⊢ ch2_:  t_
⊢ ch1_:  t_
--------------------
⊢ B (ch1_,ch2_):  t_
```

```
***** reduceC *****
⊢ c:  t_
--------------------
⊢ C c:  t_
```

The remaining rules deal with existence conditions which may arise.

```
***** reduceTyTuple *****
⊢ E t2_
⊢ E t1_
--------------------
⊢ E (TyTuple (t1_,t2_))
```

```
***** reduceType *****
⊢ E n
--------------------
⊢ E (Type n)
```

```
***** reduceSn *****
⊢ E n
--------------------
⊢ E (S n)
```

```
***** reduce0 *****
--------------------
⊢ E 0
```

C.3 Alternative Recursion Rules

The operational semantics rules dealing with the recursion in practice are different from those listed above – see Section 5.1.4. Using the auxiliary function `suspend` we modify the rules so that unification can decide for us when to apply the fix rule, and when to apply the iterate rule.

```
fun suspend l circ c (d,e) = if ceq c d then (d,e)
                                        else iterate l circ d;
```

`reducePrefix` contains the initial computation common to `reduceFix` and `reduce-Iterate`.

```
***** reducePrefix *****
1:  (initial_ ::  env_ ⊢ circ1_ ⇒ (o1_,circ1'_) :  t1_)
⊢ (initial_, o1_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_) :  t1_)
⊢ (initial_ ::  env_ ⊢ circ1_ ⇒ (o1_,circ1'_) :  t1_)
--------------------
⊢ (initial_, env_ ⊢ circ1_ ⇒ (o2_,circ2'_) :  t1_)
```

The computation of the first premise is passed on to the second premise in the hypothesis list. The hypothesis of the `reduceFix'` rule requires that the first and second approximations are equal, or strictly speaking unifiable, in the rule reduceFix' is applied to. Thus reduceFix' is applicable only if we have a fixed point.

```
***** reduceFix' *****
--------------------
```
1: $(initial\_ :: \; env\_ \vdash circ1\_ \Rightarrow (initial\_, circ1'\_) : \; t1\_)$

$\vdash (initial\_, \; initial\_, \; env\_ \vdash circ1\_ \Rightarrow (initial\_, circ1'\_) : \; t1\_)$

reduceIterate' is always applicable, and must therefore be tried after reduceFix' in any tactics.

```
***** reduceIterate' *****
```
$\vdash$ ceq $initial\_ \; o1\_$ == false

$\vdash (o1\_, \; env\_ \vdash circ1\_ \Rightarrow (o2\_, circ2'\_) : \; t1\_)$

```
--------------------
```
1: $(initial\_ :: \; env\_ \vdash circ1\_ \Rightarrow (o1\_, circ1'\_) : \; t1\_)$

$\vdash (initial\_, \; o1\_, \; env\_ \vdash circ1\_ \Rightarrow (o2\_, circ2'\_) : \; t1\_)$

It might be useful to derive reduceIterate'' which is the composition of reduceIterate' and reducePrefix because that is the only thing that can happen after reduceIterate'. Similarly we could derive reduceLetRec' which is reduceLetRec followed by reducePrefix.

C.4 Correspondence Between Paper and Embedded Operational Semantics Rules

Below we present a table with the correspondence of paper and embedded rules of the dynamic semantics. Note that there is no exact 1-1 correspondence: there is a note for the cases where no corresponding rule is present.

Construct	Embedded Rule (`reduce`-)	Paper rule
`TYPE`	**1**	3.25
`INPUT`	**2**	3.26
nil instream	`SeqNil`	3.23
h::t instream	`SeqCons`	3.24
`LET`	`Let`	3.27
`LET`	`Let`	3.35
`LET REC`	`LetRec`	3.35
`LET REC`	`Fix`, `Fix'`	3.28
`LET REC`	`Iterate`, `Iterate'`	3.29
const	`Const`	**3**
cname	`Cons`	3.36
(*const*,*const*)	`CoTuple`	3.37
?tname	`Cons`	3.38
?tname	**4**	3.39
name	`VarSn`	3.30
name	`Var0`	3.30
$e[1]$	`Index1`	3.31
$e[2]$	`Index2`	3.31
(e,e')	`Tuple`	3.32
`DELAY`	`Delay`	3.33
`IF`	`If`, `If'`, `IfTt`, `IfFf`, `IfUu`	3.34

1 The `TYPE` construct is wholly absent; see Section 4.2.1 for a justification.

2 The `INPUT` construct is not present; see Sections 4.2.1 and 4.2.2 for more information. The rule `reduceSeqCons` combines the functionality of 'paper rules' 3.26 and 3.24.

3 In the paper semantics we do not duplicate rule 3.36 for the use of *cname* as an expression. We implicitly coerce a constant to an expression. In the embedding this coercion is explicit through the `Const` constructor.

4 There is no rule for the undefined value of a constant tuple type because they cannot be created due to the omission of constant type declarations (**1** above.)

C.5 Goal Directed Use of Operational Semantics Rules

When the embedded operational semantics rules are used in a backward proof strategy, it is often useful to retain information which we computed earlier in the derivation. Consider the rule `reduceIndex1`.

```
[2] E env_, E circ_, E t1_ ⊢ E t2_
[1] ⊢ (env_ ⊢ circ_ ⇒ (CoTuple (o1_,o2_),circ'_):  TyTuple (t1_,t2_))
--------------------
⊢ (env_ ⊢ circ_[1] ⇒ (o1_,circ'_[1]):  t1_)
```

Here we have shown the existence hypotheses for the second premise, which the pretty printer omits. Strictly speaking these hypotheses are redundant, but they are often useful when dealing with circuits which are parametrised on the types *t1_* and *t2_*. If both are equal to `Type` n say, we can discharge `[2]` using the hypotheses. We could not do this if they were absent, because we cannot prove that n exists.

This optimisation becomes a liability when we use the rules in a forward proof. The reason is that we build a derivation for a type *t2_*, which is used later as part of a larger proof. The presence of the expressions *env_*, *circ_*, and *t1_* means that when we use the derivation later on, they must be instantiated with the appropriate terms. However, these terms are extraneous to the derivation of *t2_*. It is very hard to anticipate what *env_*, *circ_*, and *t1_* are required.

The solution is simple; the extra terms are removed, and a separate set of rules is provided for forward rule application purposes. The following alternative rules rules are supplied: `reduceIndex1f`, `reduceIndex2f`, `reduceLetf`, `reduceLetRecf`, `reduceIff`, `reduceIf'f`, `reduceIfFff`, `reduceIfTtf`, and `reduceIfUuf`. The rules dealing with the `IF` statement still have a problem with the subgoal which computes the output.

All embedded operational semantics rules which have more than one premise must combine more than one derivation. This corresponds to popping an element off the goal stack for every subgoal. This is accomplished by `genMergeProofTrees`, described in Sections 4.1.2 and 5.2.2. For example, `reduceIndex2f` may be derived from `reduceIndex2` as follows.

```
pushRule opsempe reduceIndex2;
(* Delete unwanted existence hypotheses:  *)
atn [2] (monoGl [1,2,3]);
val reduceIndex2f = popGoal ();
val fReduceIndex2 = mergeProofTrees false reduceIndex2f;
```

`fReduceIndex2` is a function which, when applied to a unit value `()`, combines the top two proof trees into a derivation for a circuit which has `Index2` as its outermost constuctor.

Bibliography

[1] Luiga Aiello, Mario Aiello, and Richard W Weyhrauch. The semantics of Pascal in LCF. Memo STAN-CS-74-447, Stanford Artificial Intelligence Laboratory, Computer Science Department, Stanford University, August 1974.

[2] C M Angelo, L Claesen, and H De Man. The formal semantics definition of a multi-rate DSP specification language in HOL. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, Leuven, Belgium, September 1992.

[3] Mario R Barbacci, Steve Grout, Gary Lindstrom, Michael P Malony, Elliot I Organick, and Don Rudisill. Ada as a hardware description language: An initial report. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 272–302, Amsterdam, 1985. North Holland.

[4] H Barringer, G Gough, T Longshaw, B Monahan, M Peim, and A Williams. Semantics and verification for boolean kernel ELLA using IO automata. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 65–90. ESPRIT CHARME, North Holland, June 1991.

[5] Howard Barringer, Graham Gough, and Brian Monahan. Operational semantics for hardware design languages. In P Prinetto and P Camurati,

editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 313–334. ESPRIT CHARME, North Holland, June 1991.

[6] Howard Barringer, Graham Gough, Brian Monahan, and Alan Williams. A semantics for Core ELLA. Deliverable D2.3b, Department of Computer Science, University of Manchester, November 1992. Formal Verification Support for ELLA, IED project 4/1/1357.

[7] Harry G Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24:437–491, 1984.

[8] David A Basin. Extracting circuits from constructive proofs. In *1991 International Workshop on Formal Verification in VLSI Design*. ACM IFIP WG 10.2, January 1991.

[9] David A Basin, Geoffrey Brown, and Miriam E Leeser. Formally verified synthesis of combinatorial CMOS circuits. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 251–260, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

[10] David A Basin and Peter Del Vecchio. Verification of combinatorial logic in Nuprl. In M Leeser and G Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical aspects*, pages 333–357. Springer Verlag, July 1989.

[11] J C Bicarregui and B Ritchie. Proving support for the formal development of software, April 1989.

[12] Mark Bickford and Mandayam Srivas. Verification of a fault-tolerant property of a multi-processor system. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 225–251. IFIP TC10/WG 10.2, North Holland, June 1992.

[13] J Bormann, H Nusser-Wehlan, and G Venzl. Formal design in an industrial research laboratory: Lessons and perspectives. In Jørgen Staunstrup and Robin Sharp, editors, *Second Workshop on Designing Correct Circuits*, pages 193–213, Lynbgy, Denmark, January 1992. IFIP WG 10.2, WG 10.5.

[14] D Borrione and J L Paillet. An approach to the formal verification of VHDL descriptions. Technical Report RR 683-I-, IMAG/ARTEMIS, November 1987.

[15] Dominique Borrione, David Deharbe, Hans Eveking, and Stefan Höreth. Applications of a BDD-package to the verification of HDL descriptions. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 385–400. ESPRIT CHARME, North Holland, June 1991.

[16] Dominique Borrione, Laurence Pierre, and Ashraf Salem. PREVAIL: A proof environment for VHDL descriptions. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 163–186. ESPRIT CHARME, North Holland, June 1991.

[17] Richard Boulton. A HOL semantics for a subset of ELLA. Technical Report 254, University of Cambridge Computer Laboratory, April 1992.

[18] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John van Tassel. Experience with embedding hardware description languages in HOL. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. IFIP TC10/WG 10.2, North Holland, June 1992.

[19] 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.

[20] Richard J Boulton. A lazy approach to fully-expansive theorem proving. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, Leuven, Belgium, September 1992.

[21] Robert S Boyer and J Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

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

[23] Bishop C Brock, Warren A Hunt, Jr, and William D Young. Introduction to a formally defined hardware description language. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 3–35. IFIP TC10/WG 10.2, North Holland, June 1992.

[24] Randal E Bryant. Can a simulator verify a circuit? In G Milne and P A Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 125–136, Amsterdam, 1985. North Holland.

[25] 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.

[26] Randal E Bryant. Formal verification of memory circuits by switch-level simulation. Technical Report CMU-CS-89-156, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, June 1989.

[27] Randal E Bryant. Verification of synchronous circuits by symbolic logic simulation. In M Leeser and G Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical aspects*, pages 14–24. Springer Verlag, July 1989.

[28] Giacomo Buonanno, Alberto Coen-Porisini, and William Fornaciari. Hardware specification using the assertion language ASTRAL. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 335–358. ESPRIT CHARME, North Holland, June 1991.

[29] R M Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–44, 1969.

[30] Holger Busch. *Hardware Design By Proven Transformations.* PhD thesis, Department of Electrical Engineering and Electronics, Brunel University of West London, Uxbridge, September 1991.

[31] Holger Busch. Transformational design in a theorem prover. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 175–196. IFIP TC10/WG 10.2, North Holland, June 1992.

[32] Albert Camilleri. Higher order logic mechanization of the CSP failure-divergence semantics. Technical Report HPL-90-194, HP Laboratories Bristol, September 1990.

[33] Albert John Camilleri. Simulating hardware specifications within a theorem-proving framework. *International Journal of Computer Aided Design*, 2:315–337, 1990.

[34] Paolo Camurati, Tiziana Margaria, and Paolo Prinetti. Use of the OTTER theorem prover for the formal verification of hardware. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*, pages 253–270, Oxford, September 1990. Springer Verlag.

[35] Paolo Camurati and Paolo Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE Computer*, pages

8–19, July 1988. Also in Formal Verification of Hardware Design M Yoeli (ed.), IEEE Computer Society Press Tutorial.

[36] Luca Cardelli. *An Algebraic Approach to Hardware Description and Verification.* PhD thesis, Department of Computer Science, University of Edinburgh, April 1982. CST-16-82.

[37] William C Carter, William H Joyner Jr, and Danier Brand. Symbolic simulation for correct machine design. In *16th Design Automation Conference*, pages 280–287, San Diego, California, June 1979. ACM/IEEE.

[38] Shiu-Kai Chin. Summary of higher-order metafunctions for synthesizing signed-binary arithmetic hardware. In *1991 International Workshop on Formal Verification in VLSI Design.* ACM IFIP WG 10.2, January 1991.

[39] Chang H Cho and James R Armstrong. VHDL semantics for behavioral test generation. In D Borrione and R Waxman, editors, *CHDL 91: 10th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 427–444. IFIP WG 10.2, North Holland, April 1991.

[40] Avra Cohn. High level proof in LCF. Internal Report CSR-35-78, Department of Computer Science, University of Edinburgh, November 1978.

[41] Avra Cohn. A proof of correctness of the VIPER microprocessor: The first level. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Boston, 1987. Kluwer Academic Publishers. Also as University of Cambridge Computer Laboratory report number 104.

[42] Avra Cohn. Correctness properties of the VIPER block model: The second level. In G Birtwistle and P A Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91, New York, 1988. Springer Verlag. Also as University of Cambridge Computer Laboratory report number 134.

[43] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[44] Avra Cohn and Mike Gordon. A mechanised proof of correctness of a simple counter. Technical Report 94, University of Cambridge Computer Laboratory, July 1986.

[45] Computer General Electronic Design, 5 Greenways Business Park, Chippenham, Wiltshire SN15 1BN. *The ELLA Language Reference Manual*, issue 4.0, 1990. ELLA is now marketed by R^3 Systems.

[46] Robert L Constable and Douglas J Howe. Nuprl as a general logic. In Piergiorgio Odifreddi, editor, *Logic and computer science*, volume 31 of *APIC studies in data processing*, pages 77–90. Academic Press, 1990.

[47] M B Davies. Mathematical equivalence in a primitive ELLA. Memorandum 4225, Royal Signals and Radar Establishment, August 1988.

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

[49] Carlos Delgado Kloos. *Semantics of Digital Circuits*, volume 285 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

[50] James R Duley and Donald L Dietmeyer. A digital system design language (DDL). *IEEE Transactions on Computers*, C-17(9):850–861, September 1968.

[51] Electronic Industries Association. *EDIF: Electronic Design Interchange format*, 1987.

[52] Bruce Elliot. An application of fixed point theory to the ELLA language. Internal Technical Report N045.90.3, Praxis Systems plc, April 1988. Provisional.

[53] Erasmi Roterodami (Desiderius Erasmus). *(In) Praise of Folly.* 1511. Ad. Donker Facsimile edition.

Of the same caliber are those who court immortal fame by writing books. Although all authors owe a great deal to me, especially those who blot their pages with pure unadulterated rubbish. For those who write a dissertation, which is only meant to be subjected to the judgement of a few professors, and who do not fear the most severe and able critics, are to be lamented rather than to be envied for their continuous self-torture. They add, change, remove, lay aside, take up, rephrase, like to show to others, keep close to their heart for nine years, and are never satisfied with the result. And their futile reward, a word of praise from a person or two, is dearly paid for — so many late nights, so much sweat and anguish, and the loss of the sweetest thing there is: their sleep. Then their health deteriorates, their looks are destroyed, they suffer partial or total blindness and poverty, become ill-tempered, are out of favour, have to forsake all pleasures, age rapidly, die prematurely, and invite other disasters. But they bear all these sacrifices to be given approval by one or two learned people.

[54] Hans Eveking. Axiomatizing hardware description languages. *International Journal of Computer Aided VLSI Design*, 2:263–280, 1990.

[55] Hans Eveking and Ulf Schellin. Register-transfer level verification in SMAX. CHARME Project Report THD-2.B.2.b-01, Technische Hochschule Darmstadt, October 1991.

[56] Ivan V Fillippenko. VHDL verification in the state delta verification system (SDVS). In *1991 International Workshop on Formal Verification in VLSI Design*, January 1991.

[57] Simon Finn and Michael P Fourman. *Logic Manual for the Lambda System.* Abstract Hardware Limited, version 3.1, May 1990.

[58] Simon Finn and Michael P Fourman. *Logic Manual for the Lambda System.* Abstract Hardware Limited, version 4.0, March 1991.

[59] Simon Finn, Michael P Fourman, Michael Francis, and Robert Harris. Formal system design – interactive synthesis based on computer-assisted reasoning. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 97–110, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

[60] R W Floyd. Assigning meanings to programs. *Proceedings of American Mathematical Society, Symposia in Applied Mathematics*, 19:19–32, 1967.

[61] Michael P Fourman and Simon Finn. *Logic Manual for the Lambda System.* Abstract Hardware Limited, version 3.0, November 1989.

[62] 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.

[63] Mick Francis. *DIALOG Reference Manual.* Abstract Hardware Limited, version 3.2, December 1990.

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

[65] Mick Francis, Simon Finn, and Ellie Mayger. *Reference Manual for the Lambda System.* Abstract Hardware Limited, version 3.1, May 1990.

[66] Masahiro Fujita. Hardware verification based on HDL sources. In R W Hartenstein, editor, *Hardware Description Languages*, volume 7 of *Advances in CAD for VLSI*, chapter 4, pages 283–312. Elsevier Science Publishers (North Holland), 1987.

[67] Sumit Ghosh. Using Ada as an HDL. *IEEE Design and Test*, pages 30–42, February 1988.

[68] Joseph A Goguen. OBJ as a theorem prover with applications to hardware verification. In G Birtwistle and P A Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 219–267, New York, 1988. Springer Verlag.

[69] K G W Goossens. Embedding computer hardware design and description languages in proof systems. Thesis Proposal, December 1989.

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

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

[72] K G W Goossens. Embedding a CHDDL in a proof system. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 359–374. ESPRIT CHARME, North Holland, June 1991. Also as LFCS Report ECS-LFCS-91-155.

[73] K G W Goossens. Operational semantics based formal symbolic simulation. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, Leuven, Belgium, September 1992. A longer version is available as LFCS Report ECS-LFCS-92-231.

[74] Ganesh Gopakakrishnan, Richard M Fujimoto, Vankatesh Akella, N S Mani, and kevin N Smith. Specification-driven design of custom hardware in HOP. In G Birtwistle and P A Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 128–170, New York, 1988. Springer Verlag.

[75] Andrew D Gordon. The formal definition of a synchronous hardware description language in higher order logic. In *International Conference on Computer Design*, October 1992.

[76] M Gordon, R Milner, and C Wadsworth. Edinburgh LCF. Internal Report CSR-11-77, Department of Computer Science, University of Edinburgh, May 1977.

[77] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[78] Michael J C Gordon. The denotational semantics of sequential machines. *Information Processing Letters*, 10(1):1–3, February 1980.

[79] Michael J C Gordon. HOL: A proof generating system for higher-order logic. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, Boston, 1987. Kluwer Academic Publishers.

[80] Mike Gordon. A model of register transfer systems with applications to microcode and VLSI correctness. Internal Report CSR-82-81, Department of Computer Science, University of Edinburgh, March 1981.

[81] Mike Gordon. LCF_LSM. Technical Report 41, University of Cambridge Computer Laboratory, September 1983. Second Printing with Corrections and Additions.

[82] Mike Gordon. Proving a computer correct. Technical Report 42, University of Cambridge Computer Laboratory, 1983. With the LCF_LSM hardware verification system.

[83] Mike Gordon. Why higher-order logic is a good formalisation for specifying and verifying hardware. In G Milne and P A Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177, Amsterdam, 1985. North Holland.

[84] F K Hanna and N Daeche. Specification and verification using higher-order logic. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 418–433, Amsterdam, 1985. North Holland.

[85] F K Hanna, N Daeche, and M Longley. Veritas+: A specification language based on type theory. In M Leeser and G Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical aspects*, pages 358–379. Springer Verlag, July 1989.

[86] F K Hanna, M Longley, and N Daeche. Formal synthesis of digital systems. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 532–548, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

[87] Keith Hanna and Neil Daeche. The Veritas design logic: A user's view. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 301–310. IFIP TC10/WG 10.2, North Holland, June 1992.

[88] Keith Hanna, Neil Daeche, and Gareth Howells. Implementation of the Veritas design logic. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 77–94. IFIP TC10/WG 10.2, North Holland, June 1992.

[89] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Symposium on Logic in Computer Science, Ithaca, NY*, June 1987.

[90] 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.

[91] John P Hayes. Digital simulation with multiple logic values. *IEEE Transactions on Computer-Aided Design*, CAD-5(2):274–283, April 1986.

[92] M G Hill. The dynamic semantics of kernel ELLA. Memorandum 4630, Defence Research Agency, Malvern, UK, August 1992.

[93] M G Hill, E V Whiting, and J D Morison. Formal semantic definition of ELLA timing. Memorandum 4436, Royal Signals and Radar Establishment, November 1990.

[94] M G Hill, E V Whiting, and J D Morison. Sprite-ELLA language enhancements. Memorandum 4441, Royal Signals and Radar Establishment, November 1990.

[95] C A R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[96] C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[97] W Howard. The formulas-as-types notion of construction. In J P Sledin and J R Hindley, editors, *To H B Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 470–490. Academic press, 1980.

[98] Warren A Hunt, Jr. FM8501: A verified microprocessor. Technical Report 47, Institute for Computing Science. The University of Texas at Austin, December 1985. Dissertation.

[99] Warren A Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[100] IEEE computer, December 1974. Special Edition on Hardware Description Languages.

[101] 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, 1988.

[102] Paul B Jackson. Nuprl and its uses in circuit design. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 311–336. IFIP TC10/WG 10.2, North Holland, June 1992.

[103] Steven D Johnson. *Synthesis of Digital Designs from Recursion Equations*. ACM Distinguished Dissertation. The MIT Press, 1983. Indiana University PhD Thesis, May 1983.

[104] Steven D Johnson. Manipulating logical organization with system factorizations. In M Leeser and G Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical aspects*, pages 260–281. Springer Verlag, July 1989.

[105] C B Jones and P A Lindsay. A support system for formal reasoning: Requirements and status. In R Bloomfield, L Marshall, and R Jones, editors, *VDM '88: VDM — The Way Ahead*, pages 139–152. Springer Verlag, September 1988. Lecture Notes in Computer Science 328.

[106] Jeff Joyce, Graham Birtwistle, and Mike Gordon. Proving a computer correct in higher order logic. Technical Report 100, University of Cambridge Computer Laboratory, December 1986. HOL version of Technical Report 42.

[107] Jeffrey J Joyce. A verified compiler for a verified microprocessor. Technical Report 167, University of Cambridge Computer Laboratory, March 1989.

[108] G Kahn. Natural semantics. Gipe project second annual review report, INRIA, Sophia-Antipolis, France, January 1987.

[109] K Khordoc, M Biotteau, and E Cerny. Switch-level models in multi-level VHDL simulations. In *Proceedings of the First European Conference on VHDL*, Marseille, September 1990. IMT.

[110] Stephen Cole Kleene. *Introduction to Metamathematics.* Bibliotheca mathematica. North Holland, Amsterdam, 1952.

[111] David C Ku and Giovanni De Micheli. Hardware C: A language for hardware design. Technical Report CSL-TR-88-362, Computer Systems Laboratory, Stanford University, August 1988.

[112] S Leinwand and T Lamdan. Design verification based on functional abstraction. In *16th Design Automation Conference*, pages 353–359, San Diego, California, June 1979. ACM/IEEE.

[113] Beth Levy, Ivan Fillipenko, Leo Markus, and Telis Menas. Using the state delta verification system (SDVS) for hardware verification. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 337–360. IFIP TC10/WG 10.2, North Holland, June 1992.

[114] Beth H Levy. An overview of the state delta verification system (SDVS). In *1991 International Workshop on Formal Verification in VLSI Design.* ACM IFIP WG 10.2, January 1991.

[115] Wayne Luk. Optimising designs by transposition. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*, pages 332–354, Oxford, September 1990. Springer Verlag.

[116] Zhaohui Luo and Robert Pollack. LEGO proof development system: Users's manual. LFCS Report Series ECS-LFCS-92-211, LFCS, Department of Computer Science, University of Edinburgh, May 1992.

[117] Jean-Christophe Madre and Jean-Paul Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 205–210, 1988. Also in Formal Verification of Hardware Design M Yoeli (ed.), IEEE Computer Society Press Tutorial.

[118] Zohar Manna, Stephen Ness, and Jean Vuillemin. Inductive methods for proving properties of programs. *ACM SIGPLAN Notices*, 7:27–50, 1972.

[119] Ian A Mason. Hoare's logic in the LF. LFCS Report Series ECS-LFCS-87-32, LFCS, Department of Computer Science, University of Edinburgh, June 1987.

[120] Michael C McFarland and Alice C Parker. An abstract model of behavior for hardware descriptions. *IEEE Transactions on Computers*, C-32(7):621–637, July 1983.

[121] Thomas F Melham. Abstraction mechanisms for hardware verification. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291, Boston, 1987. Kluwer Academic Publishers.

[122] Thomas F Melham. Automating recursive type definitions in higher order logic. In G Birtwistle and P A Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386, New York, 1988. Springer Verlag.

[123] Thomas F Melham. Using recursive types to reason about hardware in higher order logic. Technical Report 135, University of Cambridge Computer Laboratory, May 1988.

[124] Thomas Frederick Melham. Formalising abstraction mechanisms for hardware verification in higher order logic. Technical Report 201, University of Cambridge Computer Laboratory, August 1990. PhD Thesis.

[125] T F Melham. A package for inductive relation definitions in HOL. In Myla Archer, Jeffrey J Joyce, Karl N Levitt, and Phillip J Windley, editors, *The HOL Theorem Proving System and Its Applications*, pages 350–357. IEEE Computer Society Press, August 1991.

[126] Meta-software Inc. *HSPICE Users' Manual H8801*, January 1988.

[127] R Milner and R Weyhauch. Proving compiler correctness in a mechanised logic. In B Meltzer and D Mitchie, editors, *Machine Intelligence*, chapter 3. Edinburgh University Press, 1972.

[128] Robin Milner. An algebraic definition of simulation between programs. Computer Science Report CS-205, Computer Science Department, Stanford University, February 1971.

[129] Robin Milner. Processes: A mathematical model of computing agents. In Rose and Shepherdson, editors, *Logic Colloquium 73: Studies in Logic and Foundations of Mathematics*, volume 80, pages 157–173. North Holland, 1973.

[130] Robin Milner. LCF: A way of doing proofs with a machine. In *8th MFCS Symposium*, Olomonc, Czechoslovakia, 1979. Springer Verlag. Lecture Notes in Computer Science.

[131] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[132] Faron Moller. The definition of CIRCAL. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 178–187, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

[133] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[134] Richard Moore. Mural: A formal development support environment. In *SafetyNet '90 Conference*, October 1990.

[135] J D Morison. Personal Communication, November 1990.

[136] J D Morison and M G Hill. A formal definition of the static semantics of ELLA's core. Technical Report 91024, Royal Signals and Radar Establishment, Malvern, Worcestershire, August 1991.

[137] J D Morison, N E Peeling, and T L Thorp. The design rationale of ELLA, a hardware design and description language. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 303–320, Amsterdam, 1985. North Holland.

[138] J D Morison, N E Peeling, and E V Whiting. Sequential programming extensions to ELLA, with automatic transformation to structure. In *International Conference on Computer Design*, 1987.

[139] Ben Moszkowski. A temporal logic for multi-level reasoning about hardware. In T Uehara and M Barbacci, editors, *CHDL 83: 6th International Symposium on Computer Hardware Description Languages and their Applications*, pages 79–90, Amsterdam, 1983. North Holland.

[140] John T O'Donnell. Hardware description with recursion equations. In M R Barbacci and C J Koomen, editors, *CHDL 87: 8th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 363–382. IFIP WG 10.2, North Holland, 1987.

[141] US Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A, January 1983.

[142] L C Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[143] Lawrence C Paulson. Natural deduction as higher-order resolution. Technical Report 82, University of Cambridge Computer Laboratory, December 1985. Revised version.

[144] N E Peeling and J D Morison. A database approach to design data management and programming support for ELLA, a high-level HDDL. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 354–363, Amsterdam, 1985. North Holland.

[145] Laurence Pierre. From a HDL description to formal proof systems: Principles and mechanization. In D Borrione and R Waxman, editors, *CHDL 91: 10th International Symposium on Computer Hardware Description Languages and Their Applications*, April 1991.

[146] Laurence Pierre. One aspect of mechanizing formal proof of hardware: The generalization of partial specifications. In *1991 International Workshop on Formal Verification in VLSI Design*. ACM IFIP WG 10.2, January 1991.

[147] R Piloty, M Barbacci, D Borrione, D Dietmeyer, F Hill, and P Skelly. *Conlan Report*, volume 151 of *Lecture Notes in Computer Science*. Springer Verlag, 1983.

[148] Robert Piloty and Dominique Borrione. The Conlan project: Concepts, implementations and applications. *IEEE Computer*, pages 81–92, February 1985.

[149] Vaijay Pitchumani and Edward P Stabler. A formal method for computer design verification. In *19th Design Automation Conference*, pages 809–814, 1982.

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

[151] Praxis Systems plc, 20 Manvers Street, Bath BA1 1PX. *The ELLA Language Reference Manual*, issue 3.0, 1986. ELLA is now marketed by R^3 Systems.

[152] Suresh Rajgopal, Kye Hedlund, and Douglas Reeves. Integrating hardware verification with design automation. In *1991 International Workshop on Formal Verification in VLSI Design*. ACM IFIP WG 10.2, January 1991.

[153] Brian Ritchie. *The Design and Implementation of an Interactive Proof Editor*. PhD thesis, Department of Computer Science, Edinburgh University, October 1988.

[154] Brian Ritchie and Paul Taylor. The interactive proof editor. An experiment in interactive theorem proving. In G Birtwistle and P A Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 303–322, New York, 1988. Springer Verlag.

[155] Lars Rossen. Formal Ruby. In *Summer School on Formal Methods for VLSI Design*, pages 163–174. IFIP WG 10.5, June 1990.

[156] Lars Rossen and Robin Sharp. Sequence semantics of ruby. In Jørgen Staunstrup and Robin Sharp, editors, *Second Workshop on Designing Correct Circuits*, pages 159–171, Lynbgy, Denmark, January 1992. IFIP WG 10.2, WG 10.5.

[157] Ashraf Salem and Dominique Borrione. Formal semantics of VDHL timing constructs. In *Euro-VHDL Stockholm*, September 1991.

[158] James B Saxe, Stephen J Garland, John V Guttag, and James J Horning. Using transformations and verification in circuit design. In Jørgen Staunstrup and Robin Sharp, editors, *Second Workshop on Designing Correct Circuits*, pages 1–25, Lynbgy, Denmark, January 1992. IFIP WG 10.2, WG 10.5.

[159] David A Schmidt. *Denotational Semantics, A Methodology for Language Development.* Allyn and Bacon Inc, Boston, 1986.

[160] Dana S Scott. *Identity and Existence in Intuitionistic Logic*, volume 753 of *Lecture Notes in Mathematics*, pages 661–696. Springer Verlag, Berlin, Heidelberg, New York, 1979. Applications of Sheaves, Proceedings, Durham 1977, eds. M.P. Fourman, C.J.Mulvey and D.S. Scott.

[161] Moe Shahdad, Roger Lipsett, Erich Marschner, Kellye Sheenan, Howard Cohen, Ron Waxman, and Dave Ackley. The VHSIC hardware description language. *IEEE Computer*, pages 94–103, February 1985.

[162] Satnam Singh. Circuit analysis by non-standard interpretation. In Jørgen Staunstrup and Robin Sharp, editors, *Second Workshop on Designing Correct Circuits*, pages 199–138, Lynbgy, Denmark, January 1992. IFIP WG 10.2, WG 10.5.

[163] Stefan Sokolowski. Soundness of Hoare's logic: An automated proof using LCF. *ACM Transactions on Programming Languages and Systems*, 9(1):100–120, January 1987.

[164] Spinoza. *Ethica.* 1665. 1979 Wereldbibliotheek edition.
Proposition 52: An object which we have seen previously together with other objects, or which in our opinion only has attributes in common with many other objects, we will not pay the same attention as an object which we imagine to have something special.

[165] J Staunstrup and M R Greenstreet. Synchronized transitions. In *Summer School on Formal Methods for VLSI Design*, pages 3–61. IFIP WG 10.5, June 1990.

[166] Jørgen Staunstrup, Stephen J Garland, and John V Guttag. Mechanised verification of circuit descriptions using the Larch prover. In V Stavridou,

T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 277–299. IFIP TC10/WG 10.2, North Holland, June 1992.

[167] V Stavridou, J A Goguen, S M Elker, and S N Aloneftis. FUNNEL: A CHDL with formal semantics. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 115–137. ESPRIT CHARME, North Holland, June 1991.

[168] V Stavridou, J A Goguen, A Stevens, S M Eker, S N Alonefits, and K M Hobley. FUNNEL and 2OBJ: Towards and integrated hardware design environment. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 197–223. IFIP TC10/WG 10.2, North Holland, June 1992.

[169] Joseph E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[170] Edward W Thompson. Simulation — A tool in an integrated testing environment. In *1980 Test Conference*, 1980.

[171] Takao Uehara, Takao Saito, Fumihiro Maruyama, and Nobuaki Kawato. DDL verifier and temporal logic. In T Uehara and M Barbacci, editors, *CHDL 83: 6th International Symposium on Computer Hardware Description Languages and their Applications*, pages 91–102, Amsterdam, 1983. North Holland.

[172] Gabriele Umbreit. Providing a VHDL-interface for proof systems. In *EURO-DAC*, pages 698–703, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1264, September 1992. IEEE Computer Society Press.

[173] Filip van Aelten and Jonathan Allen. Efficient verification of VLSI circuits based on syntax and denotational semantics. In Luc Claesen, editor,

Applied Formal Methods For Correct VLSI Design, pages 188–197, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

[174] John van Tassel and David Hemmendinger. Toward formal verification of VHDL specifications. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 261–270, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

[175] John P van Tassel. A formalisation of the VHDL simulation cycle. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, Leuven, Belgium, September 1992.

[176] John Peter van Tassel. The semantics of VHDL with VAL and HOL: Towards practical verification tools. Technical Report 196, University of Cambridge Computer Laboratory, 1990. MSc Thesis of Wright State University.

[177] Li-Guo Wang. Hardware synthesis logic and its independence. Manuscript. Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, November 1991.

[178] Daniel Weise. Functional verification of MOS circuits. In *24th Design Automation Conference*, pages 265–270, Miami Beach, Florida, 1987. ACM/IEEE.

[179] Daniel Weise. Constraints, abstraction, and verification. In M Leeser and G Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical aspects*, pages 25–39. Springer Verlag, July 1989.

[180] Alan Williams. Verification requirements analysis. Technical Report Deliverable D2.1, University of Manchester, September 1990. Formal Verification Support for ELLA IED 4/1/1357.

[181] Philip A Wilsey. Developing a formal semantic definition of VHDL. In *Proceedings of the First European Conference on VHDL*, Marseille, September 1990. IMT.

[182] Philip A Wilsey, Timothy J McBrayer, and David Sims. Towards a formal model of VLSI systems compatible with VHDL. In A Halaas and P B Denyer, editors, *VLSI '91*, pages 6a.2.1–6a.2.12, Edinburgh, Scotland, August 1991. IFIP TC 10/WG 10.5.

[183] Phillip J Windley. Abstract hardware. In *1991 International Workshop on Formal Verification in VLSI Design*. ACM IFIP WG 10.2, January 1991.

[184] Phillip J Windley. Abstract theories in HOL. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, Leuven, Belgium, September 1992.

[185] Glynn Winskel. Models and logic of MOS circuits. In M Broy, editor, *International Summer School on Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series*. Springer Verlag, July–August 1986. Also as University of Cambridge Computing Laboratory technical report number 96.

[186] Ching-Farn E Wu, Anthony S Wojcik, and Lionel M Ni. A rule-based circuit representation for automated CMOS design and verification. In *24th Design Automation Conference*, pages 786–792, Miami Beach, Florida, 1987. ACM/IEEE.

[187] William D Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5:493–518, 1989.

[188] Zheng Zhu and Steven D Johnson. An example of interactive hardware transformation. In *1991 International Workshop on Formal Verification in VLSI Design*. ACM IFIP WG 10.2, January 1991.

Stelling 52: Een voorwerp dat wij reeds vroeger gelijktijdig met anderen hebben gezien of dat naar onze voorstelling uitsluitend eigenschappen bezit, die het gemeen heeft met vele andere voorwerpen, zullen wij niet zolang onze aandacht schenken als een waarvan wij ons voorstellen dat het iets bijzonders heeft.

Spinoza

Ethica [164, p 171]