variable values for signals and not expressed their interrelationships. A conditional split must then be introduced into the behaviour of a program to allow for both possibilities (i.e. where one branch assumes an event and the other does not). The user may of course choose at this stage to specialise the variables being used to ensure either one of these possible behaviours. If not, `finish_GAMMA` introduces the appropriate split. Examples of the use of `finish_GAMMA` in context are provided in Chapter 6.

The use of `finish_GAMMA` emphasises one particularly troublesome feature of the animation of not only Femto-VHDL programs, but also of full VHDL. If care is not taken in the specification of initial values of signals, it is quite possible to engender this kind of conditional split for every signal almost every time around the simulation loop. The result is naturally an explosion in the size of the terms being examined. It also puts one in the situation of doing exhaustive search over all possible values for the signals, which is exactly the kind of reasoning that is trying to be avoided.

From the point of view of automation, the case splits engendered by `finish_GAMMA` are one of the reasons behind not attempting to construct a completely automatic tool. Such choice points may be of interest in individual circumstances where one particular course of action needs to be taken. If, however, one was not interested in human intervention, it would be simple to write a naïve conversion to repetitively go around the simulation loop simply by composing `ONCE_AROUND` and `finish_GAMMA`. The problem is, of course, that the simulation may not quiesce. In those situations, the new conversion would loop infinitely.

The three parts of the cycle, "move to nearest interesting point", "update current state" and "calculate events", are animated by `simp_state_CONV`. The conversion first traverses the list of delays trying to determine the next point of computation. Having ascertained that time, the posted transactions for that time are extracted. These transactions are then compared to the existing signal values to give a new set of signal-value pairs and a new set of events via `NEXT_SIGMA_CONV` and `NEXT_GAMMA_CONV` respectively. Once the new environment has been derived in this fashion, the final part of the diagram ("perform a simulation cycle") is entered into. Here `RUNCYCLE_CONV` is the top level for another set of related conversions which execute the simulation cycle.

Also recall the steps that make up a simulation cycle:

- determine which processes are active based on current events
- run the active processes in parallel
- merge all processes' scheduled transactions into a collective whole

`RUNCYCLE_CONV` first traverses the sea of processes to wake up and run any of them which are sensitive to the just-derived events. This is accomplished by making use of the rules for the simulation cycle as defined in Chapter 4 and embedded in the HOL system via `new_inductive_definition`. Once a process has been activated, the statements inside it are animated by use of `RUNSEQ_CONV` and `RUNBOOL_CONV`, which are responsible for the execution of the rules for sequential statements and Boolean expressions respectively. The behaviour resulting from the activation of any two processes in parallel is resolved from calls to clean_zip to zip by `CLEAN_ZIP_CONV`.

This account of the sequence of conversions hides the fact that simplification of the results arising from many of the steps is also being carried out. A simple example is that calls to clean_zip are reduced to the appropriate sequence of zip's. These calls to zip are also further simplified by the collapsing out of any empty transactions ($\lambda t.\{\}$). Once normalised, the resulting transactions are more concise and therefore make the work of `simp_state_CONV` much easier.

The Conversion `finish_GAMMA`

It is sometimes the case that when `NEXT_GAMMA_CONV` is attempting to ascertain the next set of events, there is not enough information about the state of the signals to ascertain whether or not a new value for particular a signal represents a change from the old one. This situation frequently occurs when the user has supplied

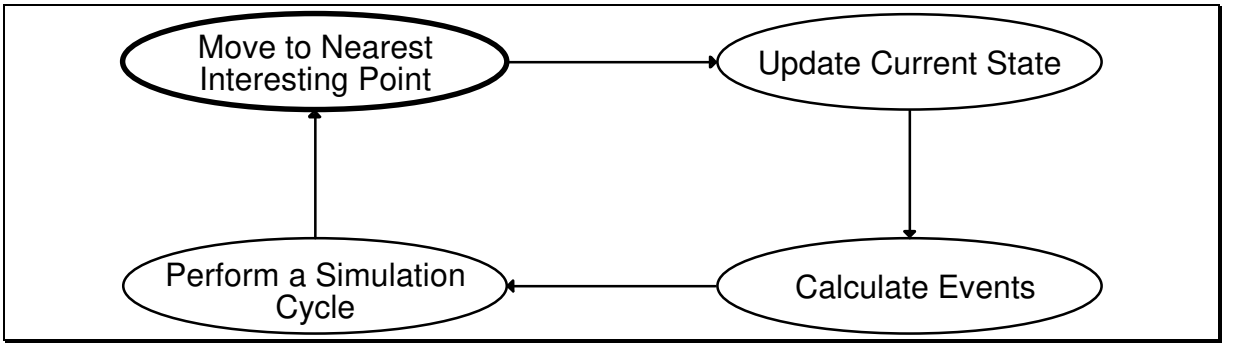
those behaviours or their equivalence to other behaviours may be discerned by proof.

In order to make use of the semantics to derive these behaviours for individual programs, it becomes necessary to write special-purpose conversions to animate the definitions. To a large extent, the process is one of transforming one theorem into another by expanding the definitions that it comprises in an intelligent way. To that end, almost every definition made in the semantics has had a HOL conversion written for it. These conversions are often rewriting engines to replace a definition with its body. Frequently however, it is possible to do some simplification of the result to get a clearer statement (simplification of arithmetic expressions, normalisation, etc.). In these cases the conversions must also do context-dependant reasoning. Most of the time, this leads to faster execution time as it cuts down on the size of the terms being manipulated.

In the course of working with the semantics of individual Femto-VHDL programs, two particular conversions will be frequently used. The first is one that animates one iteration of the simulation loop, and is aptly-enough named `ONCE_AROUND`. The other, `finish_GAMMA`, helps trigger an event when there is insufficient contextual information about signal values. None of the ML code for either of these conversions will be given. Instead the algorithm and rationale behind each will be presented. Examples of their use follow in Chapter 6.

The Conversion `ONCE_AROUND`

Recall the diagram for the top-level simulation loop:



The purpose of `ONCE_AROUND` is to co-ordinate these steps for individual Femto-VHDL programs by using the rules and definitions of the semantics. Each of the actions above represents a suite of additional proof procedures which animate the related definitions.

When given to the defining function, they appear as:

```
let (csrules,csind) =
    let CS="--cyc-->:cycenv->CS->trans->trans->bool" in
    new_inductive_definition false `--cyc-->_DEF`
    ("^CS ρ Stmnt τ τ'",[])
    [[
      %--------------------------------% "(sl ∩ γ_of_ρ ρ))={}"],
      "^CS ρ (PROCESS sl SS) τ (λt.{})";
     [    "--seq--> ρ SS τ τ'";
      %---------------------------% "~((sl ∩ (γ_of_ρ ρ))={})"],
      "^CS ρ (PROCESS sl SS) τ τ'";
     [    "^CS ρ S0 τ τ'" ; "^CS ρ S1 τ τ''"
      %-----------------------------------------% ],
      "^CS ρ (PAR S0 S1) τ (CLEAN_ZIP τ τ' τ'')"]
```

The function γ_of_ρ extracts the γ portion of the environment ρ. The actual constants used for $\xrightarrow{cyc}$ and $\xrightarrow{seq}$ are `SimCycle` and `SeqStmnt` respectively. The arguments to `new_inductive_definition` are much the same as other similar functions in the HOL system. The first argument is a flag to tell the system whether or not the constant being defined is infix or prefix. The second argument is the name associated with the definition in the theory. The third argument is a 'pattern' that supplies information which is needed because the function may be used to define classes of inductively defined relations, rather than just single instances of these relations [25]. The final argument is a list of term list-term pairs, each element of which describes one rule. The term list contains the hypotheses and side conditions. An empty list is used for axioms. The term is the conclusion of the rule. In HOL, any text between pairs of `%` is interpreted as a comment. So, each of the lines separating hypotheses from conclusion are really nothing more than comments inserted for readability. The result of making an inductive definition in HOL is a suite of theorems comprising both rewriting rules for each rule of the semantics and a rule induction scheme.

5.2.2 Animation

Why bother with the animation of Femto-VHDL inside HOL at all? First, it reinforces one's confidence that the semantics is actually expressing the intended intuition [13]. The deficiency in understanding which led to the replacement of zip by clean_zip in the rules for concurrent statements was exposed in just this way. Second, it is the only robust way of deriving the behaviour of specific programs so that properties of

instances of the definition into the actions that are required. All the other simple (i.e. non-recursive or inductive) definitions presented in Chapters 3 and 4 were made in an identical manner.

Recursive definitions require that the function `new_recursive_definition` be used to express them. An example of its use can be seen with the realisation of dflat in HOL. The function was given in Chapter 3 as:

dflat *design* ≡
  if *design* = arch *ent arch ports decls cs* then *cs*
  else let (dpar *d′ d″*) = *design* in (dflat *d′*) || (dflat *d″*)

and may be embedded in HOL by:

```
let DFLAT = new_recursive_definition false DESIGN `DFLAT`
    "(DFLAT (ARCH ent arch ports decls cs) = cs) /\
     (DFLAT (DPAR d' d'') = PAR (DFLAT d') (DFLAT d''))"
```

The arguments to the function are a theorem describing the recursion scheme, a flag stating if the constant being defined (in this case `DFLAT`) is prefix or infix, the name the definition will be stored under in the theory, and a term expressing the desired function as a conjunction of recursion equations establishing its behaviour. The recursion theorem is a by-product of defining the type being recursed over [28]. In this case it is `DESIGN`, which was just defined as a part of the syntax of Femto-VHDL. Like `new_definition`, `new_recursive_definition` returns as a theorem a conjunction that looks exactly like the term used in the definition. Each part of the conjunction may then be used independently to rewrite instances of `DFLAT`.

The rules for the semantics make use of the inductive definition mechanism provided in HOL [25]. The rules were transcribed from those of Chapter 4 into the syntax required by the package. Recall the rules for concurrent statements:

$cs \rightarrow transactions \rightarrow transactions \rightarrow Boolean$

$$\{\mathsf{cs}_1\}\quad \frac{}{(t,\sigma,\gamma,\theta) \vdash \langle sl : ss, \tau\rangle \xrightarrow{cyc} \lambda t.\{\}} \quad (sl \cap \gamma) = \{\}$$

$$\{\mathsf{cs}_2\}\quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle ss, \tau\rangle \xrightarrow{seq} \tau'}{(t,\sigma,\gamma,\theta) \vdash \langle sl : ss, \tau\rangle \xrightarrow{cyc} \tau'} \quad (sl \cap \gamma) \neq \{\}$$

$$\{\mathsf{cs}_3\}\quad \frac{\rho \vdash \langle cs', \tau\rangle \xrightarrow{cyc} \tau' \quad \rho \vdash \langle cs'', \tau\rangle \xrightarrow{cyc} \tau''}{\rho \vdash \langle cs' \,||\, cs'', \tau\rangle \xrightarrow{cyc} \mathsf{clean_zip}\ \tau\ \tau'\ \tau''}$$

terms called DESIGN. As an example, the following shows the HOL definition of this type:

```
let DESIGN = define_type `DESIGN`
 (`DESIGN = ARCH string string `^
                `((string)set#(string)set#(string)set) DECLS CS `^
         `| DPAR DESIGN DESIGN`)
```

The definition closely resembles the BNF specification of the syntax of designs that was made in Chapter 3. The function which defines the type takes two strings as arguments. The first is the name under which the definition will be stored in the theory. The second is a grammar giving the structure of the desired type. The symbol ^ is the ML string concatenation operator, and pairs of ` delineate those strings. The definition makes use of other types constructed in an identical manner. One of them is for local declarations (DECLS), the other for concurrent statements (CS).

Semantics

The definitions which make up the semantics of Femto-VHDL are just as straightforward as those which embed its syntax. Three different functions were used in this process. The first makes non-recursive definitions, the second makes recursive ones and the third allows inductive ones to be made. Examples of each are given below.

The simplest form of definition may be illustrated by using it to set up the equivalence relation for events ($\overset{\gamma}{=}$). Recall its definition from Chapter 4:

$$\gamma' \underset{sigs}{\overset{\gamma}{=}} \gamma'' \equiv \left(\{s : s \in \gamma' \wedge s \in sigs\} = \{s : s \in \gamma'' \wedge s \in sigs\}\right)$$

When put into HOL, the following definition is made:

```
let EQ_GAMMA = new_definition (`EQ_GAMMA`,
    "EQ_GAMMA (gam':events) (gam'':events) sigs =
       {x|(x IN gam')/\(x IN sigs)}={x|(x IN gam'')/\(x IN sigs)}
```

The defining function takes a pair as its argument. The first element is a string giving the name that the definition will be stored under in the theory. The second is a term expressing the desired functionality. Free (unbound) variables in the term will cause new_definition to fail. The resulting theorem, which is in the same form as the term given to new_definition, may subsequently be used to rewrite

The system also provides many tools that make the task of proof easier. Most of the ones necessary for our purposes have been described above. The only one not covered has been the type definition tool [28]. The tool allows the user to define recursive types in the logic. Heavy use will be made of it in expressing the syntax of Femto-VHDL, as it will allow the BNF grammar that describes it to be directly embedded in the logic. The result will be that one is able to have Femto-VHDL text represented as HOL terms. Furthermore, one will also be able to define recursive functions over their syntax.

5.2 Femto-VHDL in HOL

The mechanisation of Femto-VHDL in HOL was to a large extent a transcription of the definitions made in Chapters 3 and 4 into the system. Once that task was accomplished, proof procedures in the form of conversions were written to animate the semantics. What follows is a general discussion of the way in which the embedding was realised and the top-level conversions that were devised.

5.2.1 Embedding

The embedding of hardware description languages in HOL has recently been an active area of research [9, 10, 38]. Throughout it all, two particular ways of performing this embedding have emerged:

- Represent the abstract syntax of the language in question by terms, then define within the logic semantic functions that assign meanings to terms.
- Only define semantic operators in the logic and arrange that the user interface parse input from language syntax directly to semantic structures, and also print semantic representations in language syntax.

The first style is known as "deep embedding", the second as "shallow embedding". These labels were first applied to embedded semantics in [9]. A deep embedding approach has been chosen for Femto-VHDL and mirrors the presentation of static and dynamic semantics presented in Chapters 3 and 4.

Syntax

In order to make a deep embedding, the syntax of Femto-VHDL first has to be expressed as a suite of recursive type definitions in the logic [28] leading to a type of

A proof in the HOL system is constructed by repeatedly applying inference rules to axioms or to previously proved theorems. Since proofs may consist of millions of steps, it is necessary to provide tools to make proof construction easier for the user. The kind of proof performed in the earlier example is known as *forward* proof, and tends to be used in the development of these tools. The idea of a forward direction arises from the act of inferring new facts from known ones.

A particular kind of proof tool known as a *conversion* is a rule that maps a term to a theorem expressing the equality of that term to some other term. An example is the rule for β-conversion:

$$(\lambda x.t_1)\,t_2 \;\mapsto\; \vdash (\lambda x.t_1)\,t_2 = t_1[t_2/x]$$

Theorems of this sort are used in HOL in a variety of contexts to justify the replacement of particular terms by semantically equivalent terms.

The ML type of conversions is `conv` (or `term → thm`). For example, `BETA_CONV` is an ML function of type `conv` (i.e. a conversion) that expresses β-conversion in HOL. It produces the appropriate equational theorem on β-redexes and fails elsewhere.

The other kind of proof in HOL is known as *backward* or *goal-directed* proof. Here, the user states the fact that needs to be proven as the main goal, and goes about breaking it up into smaller sub-goals that are easier to prove. The system keeps track of these sub-goals, and uses the reasoning that proved them to construct a proof of the original goal using forward proof.

5.1.7 Libraries and Tools

The HOL system provides a variety of libraries that may be combined to make proof development easier. These libraries are either application-specific proof tools, a collection of theorems encapsulating a particular mathematical domain, or a combination of the two. All libraries may be loaded dynamically into a running copy of the HOL system. For purposes of the material presented in this document, use has been made of the following libraries:

- `arith`: Suite of proof procedures for linear arithmetic [8].
- `ind_defs`: Tool for making inductive definitions [25].
- `string`: Logical types for ASCII character codes and strings [27].
- `sets`: Theory of finite and infinite sets [26].

Theorems are represented in HOL by values of type `thm`. The only way to create theorems is by generating a proof. In HOL, this consists in applying ML functions representing *rules of inference* to axioms or previously generated theorems. The sequence of such applications directly corresponds to a logician's proof.

There are five axioms of the HOL logic and eight primitive inference rules. The axioms are bound to ML names. For example, the Law of Excluded Middle is bound to the ML name `BOOL_CASES_AX`, and looks like `|- ∀t. (t = T) ∨ (t = F).`

Theorems are printed with a preceding turnstile |– as illustrated above. Rules of inference are ML functions that return values of type `thm`. An example of a rule of inference is *specialisation* (or ∀-elimination). In standard 'natural deduction' notation this is:

$$\frac{\Gamma \mid\!- \forall x.\, t}{\Gamma \mid\!- t[t'/x]}$$

- $t[t'/x]$ denotes the result of substituting t' for free occurrences of x in t, with the restriction that no free variables in t' become bound after substitution.

This rule is represented in ML by a function `SPEC`, which takes as arguments a term `"a"` and a theorem `|- ∀x. t[x]` and returns the theorem `|- t[a]`, the result of substituting a for x in $t\,[x]$.

A proof using the axiom and inference rule could be constructed as follows:

1. $\mid\!- \forall t.\, t = \mathsf{T} \vee t = \bot$	[Axiom `BOOL_CASES_AX`]
2. $\mid\!- (1 = 2) = \mathsf{T} \vee (1 = 2) = \bot$	[Specialising line 1 using `SPEC` to '1 = 2']

This session consists of a proof of two steps: using an axiom and applying the rule `SPEC`.

The general form of a theorem is $t_1,\ldots,t_n \mid\!- t$, where $t_1,\ldots,t_n$ are Boolean terms called the *assumptions* and t is a Boolean term called the *conclusion*. Such a theorem asserts that if its assumptions are true then so is its conclusion. Its truth conditions are thus the same as those for the single term $(t_1 \wedge \ldots \wedge t_n) \supset t$. Theorems with no assumptions are printed out in the form $\mid\!- t$.

The five axioms and eight primitive inference rules of the HOL logic are described in detail [19]. Every value of type `thm` in the HOL system can be obtained by repeatedly applying primitive inference rules to axioms. When the HOL system is built, the eight primitive rules of inference are defined and the five axioms are bound to their ML names, all other predefined theorems are proved using rules of inference as the system is made.

that only well-formed theories can be constructed by allowing theorems to be created by *formal proof* only.

5.1.5 Definitions

In HOL, *definitions* are a special kind of axiom that are guaranteed to be consistent. The commonest (but not only) form of a definition is:

$$f\ x_1 \ldots x_n = t$$

where f is declared to be a new constant satisfying this equation (and t is a term whose free variables are included in the set x_1 … x_n). Such definitions cannot be recursive because, for example:

$$f\ x = (f\ x) + 1$$

would imply 0=1 (subtract $f\ x$ from both sides) and is therefore inconsistent.

The use of axioms carries considerable danger in general because it is very easy to assert inconsistent axioms. It is thus safer to use only definitions. At first sight this might appear impossible, but in fact all of ordinary mathematics can be developed from logic by definition alone. A theory containing only definitions is called a *definitional theory*. Many useful definitional theories are built into the HOL system, or available as libraries. Examples include theories of numbers (both natural numbers and integers), sets, bags, finite trees, group theory, properties of fixed points and more. The semantics of Femto-VHDL as embedded in HOL also represents a suite of definitional theories.

The theory of numbers built into HOL is a definitional theory that defines numbers logically. Peano's postulates are proved from the definitions of the type `num` and the constants `0` and `SUC`. It follows from Peano's postulates that certain kinds of recursion equations are equivalent to non-recursive definitions. There is a built-in theory of primitive recursion that supports this, together with tools for automatically transforming recursion equations into definitions.

5.1.6 Proof

For a logician, a formal proof is a sequence, each of whose elements is either an *axiom* or follows from earlier members of the sequence by a *rule of inference*. A theorem is the last element of a proof.

Functions have types of the form $\sigma_1 \to \sigma_2$, where σ_1 and σ_2 are the types of the domain and range of the function, respectively. The types of constants are declared in *theories*. An application $t_1\, t_2$ is badly typed if t_1 is not a function or if it is a function, but t_2 is not in its range. Lambda-terms, or λ-terms, denote functions. The symbol '\' is used as an ASCII approximation to λ. Thus '`\x. t`' should be read as '$\lambda x.\, t$'. For example, `"\x. x+1"` is a term that denotes the function $x \mapsto x+1$.

The HOL quotation parser also accepts `let`-terms superficially similar to those in ML. `let`-terms are actually abbreviations for ordinary terms which are specially supported by the parser and pretty printer. The constant `LET` is defined by the λ-term `(\f x. f x)`, and is used to encode `let`-terms in the logic. The system parser repeatedly applies the transformations:

`"let` $f\ v_1 \ldots v_n = t_1$ `in` t_2`"`	$\to$	`"LET(\`$f.t_2$`)(\`$v_1 \ldots v_n.t_1$`)"`
`"let (`v_1`,...,`v_n`) =` t_1 `in` t_2`"`	$\to$	`"LET(\(`v_1`,...,`v_n`).`t_2`)`t_1`"`
`"let` $v_1 = t_1$ `and ... and` $v_n = t_n$ `in` t`"`	$\to$	`"LET(...(LET(LET(\`$v_1 \ldots v_n.t$`)`t_1`)`$t_2 \ldots$`)`t_n`"`

5.1.4 Theories

The result of a session with the HOL system is an object called a *theory*. This object is closely related to what a logician would call a theory, but there are some differences arising from the needs of mechanical proof. A HOL theory, like a logician's theory, contains sets of types, constants, definitions and axioms. In addition, however, a HOL theory contains an explicit list of theorems that have been proved from the axioms and definitions. Logicians normally do not need to distinguish theorems that have actually been proved from those that could be proved, hence they do not normally consider sets of proven theorems as part of a theory; rather, they take the theorems of a theory to be the (often infinite) set of all consequences of the axioms and definitions. Another difference between logicians' theories and HOL theories is that, for logicians, theories are relatively static objects, but in HOL they can be thought of as potentially extendible. For example, the HOL system provides tools for adding to theories and combining theories. A typical interaction with HOL consists in combining some existing theories, making some definitions, proving some theorems and then saving the resulting new theory.

The purpose of the HOL system is to provide tools to enable well-formed theories to be constructed. All the theorems of such theories are logical consequences of the definitions and axioms of the theory. The HOL system ensures

5.1.2 The Logic

This section introduces the logic used by the HOL system. The system supports *higher order logic*, which is a version of predicate calculus with three main extensions:

- Variables can range over functions and predicates.
- The logic is *typed*.
- There is no separate syntactic category of *formulae*.

Much of the material here is taken from [19] and [18], and is presented in an informal way. The interested reader is referred to [19] for a formal set-theoretic semantics of the logic.

It is assumed the reader is familiar with predicate logic. The table below summarises the notation used.

Kind of term	*HOL notation*	*Standard notation*	*Description*
Truth, Falsity	`T`, `F`	$\top$, $\bot$	*true, false*
Negation	`~`t	$\neg t$	*not* t
Disjunction	t_1 `\/` t_2	$t_1 \vee t_2$	t_1 *or* t_2
Conjunction	t_1 `/\` t_2	$t_1 \wedge t_2$	t_1 *and* t_2
Implication	t_1 `==>` t_2	$t_1 \supset t_2$	t_1 *implies* t_2
Equality	t_1 `=` t_2	$t_1 = t_2$	t_1 *equals* t_2
$\forall$-quantification	`!`x `.` t	$\forall x.\, t$	*for all* x: t
$\exists$-quantification	`?`x `.` t	$\exists x.\, t$	*for some* x: t
ε-term	`@`x `.` t	$\varepsilon x.\, t$	*an* x *such that*: t
Conditional	`(`t `=>` t_1 `|` t_2`)`	$(t \Rightarrow t_1, t_2)$	*if* t *then* t_1 *else* t_2

5.1.3 Terms

Terms of the HOL logic are represented in ML by a type called `term`. They are normally input between quotation marks. As an example, the expression `"x /\ y ==> z"` evaluates to a term representing $x \wedge y \supset z$. Terms may be manipulated by various built-in functions to either extract subterms or combine them into larger terms.

Terms are quite similar to ML expressions and this can at first be confusing. Indeed, terms of the logic have types similar to ML expressions. For example, `"(1,2)"` is an ML expression with ML type `term`. The HOL type of this term is `num`$\times$`num`. By contrast, the ML expression `("1","2")` has type `term`$\times$`term`.

CHAPTER 5

MECHANISING FEMTO-VHDL

The semantics of Femto-VHDL as presented in Chapters 3 and 4 may be embedded in a mechanical proof assistant to provide ways of reasoning about programs. The proof environment that will be used is first presented. The discussion will then turn to how the mechanisation of Femto-VHDL is accomplished in this system.

5.1 HOL

The Cambridge Higher-Order Logic (HOL) system was chosen, because of its generality, as the mechanical proof development system for the embedding Femto-VHDL. Much of what follows is a paraphrase of material found in [19] and [18]. The intention is not to give an in-depth view of the system, but to provide the reader with sufficient understanding to comprehend the material that is to be presented later.

5.1.1 ML

The HOL system is built upon an implementation of the language ML, or "Meta Language". ML is an interactive language. At top-level one can evaluate expressions and perform declarations. A complete description of the syntax and semantics of ML is given in [19].

In general to evaluate an expression e one types e followed by a carriage return; the system then prints e's value and type (the type prefaced by a colon). The declaration `let` $x = e$ evaluates e and binds the resulting value to x. To bind the variables $x_1 \ldots x_n$ simultaneously to the values $e_1 \ldots e_n$ one can perform either the declaration `let` $x_1 = e_1$ `and` $\ldots$ `and` $x_n = e_n$ or `let` $x_1, \ldots, x_n = e_1, \ldots, e_n$. These two declarations are equivalent. A declaration d can be made local to the evaluation of an expression e by evaluating the expression d `in` e.