HML: AN INNOVATIVE HARDWARE DESCRIPTION LANGUAGE AND ITS TRANSLATION TO VHDL

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Yanbing Li

August 1995

© Yanbing Li 1995
ALL RIGHTS RESERVED

ABSTRACT

HML (Hardware ML) is an innovative hardware description language based on the functional programming language SML (Standard ML).

HML is a high-order language that supports polymorphic functions. HML's advanced type system provides many constructs that are important to hardware descriptions but are lacking in the more mature, widely used hardware description languages such as VHDL and Verilog. Its succinct syntax provides a simple and concise notion for describing hardware.

HML programs do not need to specify types and interfaces while describing hardware; they are automatically inferred using advanced type checking and type inference techniques.

We have implemented an HML type checker and a translator to VHDL. The HML-to-VHDL translator automatically infers types and interfaces and generates a synthesizable subset of VHDL. This makes it possible for users to describe hardware in HML and still be able to make use of the rich availability of VHDL tools or to integrate with other VHDL packages or programs.

Examples are given to illustrate the HML system.

Table of Contents

1 Introduction **1**

1.1 Purpose 2
1.2 Comparison of HML to other HDLs 2
1.3 Preview 5

2 HML Language **7**

2.1 Introduction 7
2.1.1 Introduction to SML 7
2.1.2 Introduction to HML 8
2.1.3 A Quick Example of HML 9
2.2 HML Types and Objects 10
2.2.1 HML Basic Types 11
2.2.2 Advanced Types 12
2.2.3 HML Objects and Object Declarations 14
2.3 HML Programming Constructs 16
2.3.1 HML Lexical Elements 16
2.3.2 Signal Assignments 16
2.3.3 Expressions and Operators 17
2.3.3.1 Operators 18
2.3.3.2 Bit-vectors Operations 19
2.3.3.3 Behavioral Operators 20
2.3.3.4 Expressions 21
2.3.4 Function and Hardware Function Declarations 22
2.4 Describing Hardware in HML 23
2.4.1 Describing Structure in HML 24
2.4.2 Describing Behavior in HML 26
2.4.3 Putting Behaviors and Structures Together 27
2.4.4 Restrictions of Describing Hardware in HML 27
2.5 Discrepancies Between HML and SML 28

3 HML Type Checking and Inference **30**

3.1 Introduction 30
3.2 Type Checking and Type Inference Rules 31
3.2.1 Definition of Types Used in the Type Checker 31

3.2.2 Environments 32
3.2.3 Type Inference Rules 32
3.3 Type Checking and Inference Algorithm 36

4 Translating HML to VHDL 39
4.1 Introduction to VHDL 39
4.1.1 VHDL Overview 39
4.1.2 The VHDL Subset Used in the HML-to-VHDL System . . . 40
4.1.3 Introduction to Mentor-Graphics VHDL Simulation/Synthesis Tools 41
4.2 HML-to-VHDL Translation Rules 42
4.2.1 Top-level Translation 42
4.2.2 Translation of Behaviors: Concurrent vs. Sequential Constructs 49
4.2.3 Translation of Signal Assignments 51
4.2.4 Adding Declarations in Translation 51
4.2.5 Translation for Simulation and Synthesis 53
4.2.6 Other Issues in the Translation 56
4.3 HML Features Not Implemented by HML-to-VHDL Translator . . . 57
4.3.1 Recursive Functions 58
4.3.2 High-order Functions 58
4.4 Conclusion 63

5 Implementation in SML 65
5.1 Organization of HML Source Programs 65
5.2 Data Structures and Functions 67

6 Examples 71
6.1 Non-restoring Integer Square-root 71
6.1.1 The Non-restoring Integer Square Root Algorithm 71
6.1.2 Describing the Integer Square Root in HML 72
6.1.3 VHDL Code and Simulation/Synthesis on Mentor-Graphics Tools 75
6.2 Adder/Subtracter ALU 79
6.2.1 Describing Adder/Subtracter ALU in HML 79
6.2.2 Generated VHDL Description of Adder/Subtracter ALU . . 79

7 Conclusions and Future Plans 85
7.1 Conclusions 85
7.2 Future Work 86

A HML Grammar 88

B HML2VHDL User's Manual 93

Bibliography **95**

List of Tables

1.1 Comparisons of HML and other hardware description languages . . 5

2.1 HML Types 14
2.2 Signal assignment: syntax and hardware meanings 17
2.3 HML Operators and precedence (in order of decreasing precedence) 18

4.1 The VHDL subset that is used in HML-to-VHDL system 41
4.2 HML-to-VHDL translation rules 43

List of Figures

2.1 Gate level circuit diagram of a full adder 9
2.2 Structural description of a 1-bit fullAdder 10
2.3 Behavioral description of a 1-bit fullAdder 10
2.4 HML description of an adder with carry output using bit-vector concatenation 20
2.5 HML programming constructs and program organization – a simplified HML abstract syntax tree 24
2.6 Example: A polymorphic array generator 25
2.7 An n-bit full adder generator based on a 1-bit full adder 26
2.8 Example: A polymorphic adder with latched output 26
2.9 Rewrite the polymorphic adder with latched output by mixing structural and behavioral descriptions 27
2.10 Example of multiple assignments to one signal 28

3.1 Definition of types: modes and pure types 32
3.2 HML type checking and type inference rules 33
3.3 Abstract syntax tree segment for expression (a+b)+(c+d)+1 . . . 37

4.1 HML-to-VHDL Top-level Translation Graph 44
4.2 Translated VHDL description for the fullAdder example – structure 46
4.3 Translated VHDL description for the fullAdder example – behavior 47
4.4 Translated VHDL description for an adder with output latch (mixture of structural and behavioral description) 48
4.5 Translated VHDL description for the simple adder sum 50
4.6 Translation of signal assignments 51
4.7 Translation of functions: Example of a 4-bit counter 54
4.8 Different translation for simulation and synthesis 55
4.9 Object renaming in HML-to-VHDL translation 58
4.10 Two high-order structural compositions 59
4.11 HML descriptions for two high-order structural compositions . . . 60
4.12 Translated VHDL descriptions for sequential compositions 61
4.13 Translated VHDL descriptions for parallel composition 62

5.1 Organization of HML source programs 66

6.1 Non-restoring integer square root Level 2 algorithm in SML 73

6.2 HML description of integer square root Level 2 algorithm 74
6.3 VHDL description of integer square root produced by VHDL-HML translator – Simulation Version 76
6.4 QuickVHDL simulation stimulus file of square root example 77
6.5 Simulation wave form of square root example 77
6.6 VHDL description of integer square root produced by VHDL-HML translator – Synthesis Version 78
6.7 HML description of an adder/subtracter ALU 80
6.8 VHDL description of the adder/subtracter ALU produced by VHDL-HML translator (Part 1, to be continued) 81
6.9 VHDL description of the adder/subtracter ALU produced by VHDL-HML translator (Part 2, continued) 82
6.10 VHDL description of the adder/subtracter ALU produced by VHDL-HML translator (Part 3, continued) 83

Chapter 1

Introduction

Hardware description languages (HDLs) are used to describe hardware for the purpose of simulation, modeling, testing, design, and documentation of digital systems. They are playing an important role in computer-aided design. With the progress of synthesis technology, the abstraction level of the description is getting higher. We present HML, which has a higher level of abstraction than most of the widely used HDLs.

HML is a functional hardware description language based on Standard ML (SML) [MTH90]. SML is a high-level functional programming language which is strongly typed and polymorphic and supports high-order functions. HML inherits many programming features of SML and takes advantage of many advanced techniques that are adopted from SML, including the rich type system, advanced type-checking and type inference. It also includes extensions for describing hardware, such as the concept of signal and the definition of hardware functions.

HML can be easily integrated with different back ends such as simulators and tools for hardware synthesis and verification.

1.1 Purpose

The major motivation for designing HML is to have a concise and powerful notation for hardware description. HML's high-order and advanced type system, which includes polymorphism and automatic type inference, makes this possible. As more designs are used in synthesis, it is also desirable to have a hardware description language suitable for synthesis. Among the existing HDLs, most of them have only a subset that is synthesizable.

HML was designed based on these considerations. Firstly, the most prominent feature of HML is that the automatic inference of types and interfaces allows users to write programs without specifying types. This makes HML programs easy to write and read. Secondly, strong type checking allows the user to find design rule violations such as bus width mismatches early in the design process. Thirdly, HML provides many constructs that are important to hardware description but are lacking in the more mature, widely accepted hardware description languages. Describing hardware at a higher level, HML is able to take advantage of many recent techniques in programming language research such as high-order functions and polymorphism that are not included in existing languages such as VHDL [IEE88], Verilog [TM91], and Ella [MPT85]. Finally, because more and more designs are aiming at synthesis, we implemented an HML-to-VHDL translator which allows users to write HML programs which generate synthesizable VHDL programs that can be simulated and synthesized on the commercially available VHDL tools or can be integrated with other VHDL programs.

1.2 Comparison of HML to other HDLs

There are many hardware description languages that are widely accepted; why do we need HML? The comparison of HML to other HDLs shows that HML has its

advantages. We have compared HML to VHDL, Verilog, Ella and Ruby.

Created to be a standard, VHDL [IEE88] has a very comprehensive syntax that combines both general computing and hardware description. It can support descriptions at different levels (behavioral, register transfer and structural). However, there is a subset of VHDL which doesn't have explicit hardware meaning and is theoretically not synthesizable.

VHDL is strongly typed and allows designer to specify their own enumerated types and subtypes. Since VHDL's type system is much more restrictive than that used in HML, many useful functions for hardware description cannot be written in VHDL. One such construct is a hardware generator that generates the same regular structure independent of the types of the ports of its submodule; the HML generator code is reusable on different types of submodules. VHDL doesn't support polymorphic functions; the user has to write a separate generator function for each type of the submodule. VHDL does not support recursive functions and high-order functions, which are very useful in describing hardware. In VHDL, all types and interfaces must be specified by the designer. In HML, types and interfaces are inferred automatically.

Another widely used hardware description language, Verilog [TM91], includes only a very limited notion of data types. Verilog does not allow user defined types or enumerated types, although it does allow subranges to be defined. Many of the design errors found by type checking strongly typed languages such as HML and VHDL will not be found in designs described using Verilog.

Ella [MPT85] is perhaps the HDL which is the most similar to HML. Ella is a functional language which supports user defined, enumerated types. However, all types must be specified by the designer - no attempt is made to infer types. Ella does support recursive functions but its typing system is not polymorphic, so regular structure generators with type inference cannot be specified with Ella.

Ruby [JS90] is a notation and design discipline intended for the development of regular integrated circuits and similar hardware and software architectures. The general idea of Ruby is that circuits and circuit components are represented by relations between the signals at their inputs and outputs. Larger circuits are assembled from components by a suite of functions, such as relational composition and various combining forms that represent regular arrays of components. In Ruby, polymorphism is supported by allowing the same development to be applied to many different designs. Ruby supports high-order functions and recursive functions. Ruby provides various compositions for hardware structure, but it doesn't provides features for describing behaviors. The compositions are represented in equations ("algebraic laws") which are succinct in format but are also very hard to read.

The above comparison focused on type systems of the four languages. In addition, HML is superior to other three languages with its succinct and simple syntax, which will be shown by the examples throughout the thesis. Table 1.1 summarizes the comparison of HML to other HDLs.

Table 1.1: Comparisons of HML and other hardware description languages

	VHDL	**Verilog**	**Ella**	**Ruby**	**HML**
Origin	IEEE Standard	Cadence Design System	Manchester University	Oxford University	Cornell University
Based on	Ada	C, Pascal			SML
Syntax complexity	Very complicated	Simple	Complicated	Abstract hard to read	Simple, emphasize functionality
Syntax verbosity	Very verbose	Verbose	Verbose	Succint	Succint
synthesizability	A subset synthesizable	Yes	Core Ella synthesizable	Too abstract	Generate synthesizable VHDL subset
Straightforward hardware meanings	Partially	Partially	Partially	No	Yes
Type inference	User specifies types	User specifies types	User specifies types	No types	Types inferred automatically
Polymorphic	No	No	No	Yes	Functions can apply to multiple types
High order	No	No	No	yes	Functions are first-class types
Tools	Rich availability	Rich availability	available	a small set	Translated to VHDL, use VHDL tools

1.3 Preview

HML was first described in a paper [OLLA93] which discussed HML features and how to use it to describe structural hardware. In a later paper [OLHA95], HML was used to describe the behavior of a non-restoring square root example. There was no HML parser and type checker available at that time.

In this masters work, I modified some old definitions of HML [OLLA93], added new features such as hardware functions, and did the implementation of the HML system. I have implemented a front-end HML parser, a type-checker which automatically infers types and interfaces and also checks for syntax and typing errors

and some design rule errors, and an HML-to-VHDL translator which translates HML programs into synthesizable VHDL programs.

This thesis is organized as follows. Chapter 2 gives a detailed description of the HML language, its programming features, syntax and informal semantics, particularly hardware meanings. Chapter 3 discusses the type checking / type inference rules and the algorithms that are used in the HML type checker. Chapter 4 covers the HML-to-VHDL translation rules, based on the informal semantics and hardware meanings of HML constructs that are described in Chapter 2. Chapter 5 addresses the actual implementation of the system in SML. It explains the organization of the source programs and the function of source programs. Chapter 6 gives two illustrating examples of HML. The programming techniques discussed in Chapter 2 are used to describe an integer square-root and an add/subtract example. Chapter 7 summarizes the work and discusses future directions.

There are 3 appendices. Appendix A is a summary of the HML grammar. Appendix B is the user's manual for using the HML2VHDL system. Appendix A and B together with Chapter 2 can serve as a thorough HML system manual. Appendix C is the list of all the signatures that are used in the system source code. Along with Chapter 5, it is a guide for reading and modifying the source code.

Chapter 2

HML Language

This chapter gives a detailed description of HML features, syntax, and an informal semantics. The introduction provides some background information on the Standard ML programming language (SML) and an overview of HML. The next two sections discuss HML's type system and programming constructs. The last section in this chapter is about how to describe hardware (including structural and behavioral descriptions) in HML and some programming techniques.

2.1 Introduction

HML is based on SML [MTH90], including extensions in the area of hardware descriptions. It is also implemented in SML of New Jersey. The first part of this section briefly introduces SML, the second part is an overview of HML.

2.1.1 Introduction to SML

SML (Standard ML) is a very high level programming language and is based upon a formal definition [MTH90] that prescribes the precise semantics of the language. SML is primarily a higher order functional language. SML is strongly typed and polymorphic. This retains much of the flexibility of typeless languages while pre-

venting most run-time type errors. SML's modules may be the most advanced of any language. The SML functor extends the usual notion of generic module. SML does support some procedural programming features, such as sequential operations, assignments, references, input/output commands and exception handling [Pau91].

There are several different implementations of SML available. One of the best and most widely used implementations is Standard ML of New Jersey (SML-NJ) [AT93]. SML-NJ produces efficient code and includes a variety of tools to support program development. Among those tools which were used with the HML system are: SML-Lex and SML-Yacc. Lex and Yacc provide the capability of reading input files and parsing them into a data structure.

2.1.2 Introduction to HML

HML adopts many advanced features from SML, including:

- Functional language, with some procedural programming features,
- high-order and polymorphic functions,
- automatic type inference, and
- concise and readable syntax.

HML inherits a subset of its syntax from the programming language SML. This includes type, value and function declarations, expressions and program control statements. HML also supports non-functional features such as sequential expressions and assignments in similar ways to SML.

In addition, HML adds important features specifically for hardware description, including the concept of *signals*, *behaviors* and *hardware functions*, *logic operators*, and *behavior* and *structure* constructors.

2.1.3 A Quick Example of HML

In this section we look at a small example of an HML description of a 1-bit full adder to get a feel for how to use the language for describing hardware. In the following sections of this chapter, we will discuss the HML type system, programming constructs and in greater detail, how to describe hardware using HML.

The 1-bit full adder has three inputs: `a`, `b` and carry input `cin`; it has two outputs: `sum` and `cout`. The formulas for calculating `sum` and `cout` are:

`sum` = $a \oplus b \oplus cin$

`cout` = $a \times b + a \times cin + b \times cin$ = $a \times b + cin(a \oplus b)$.

The gate level design according the the above formulas is shown in Figure 2.1.

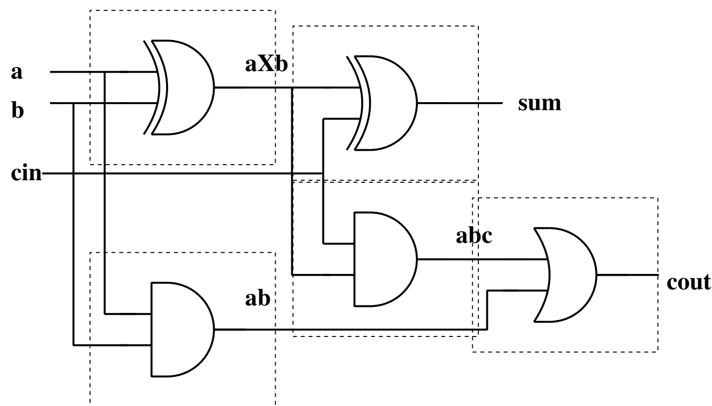

Figure 2.1: Gate level circuit diagram of a full adder

HML can describe hardware in structural and behavioral formats. Figure 2.2 gives the description of the 1-bit full adder in the structural format. **hw** is the keyword for defining a hardware function. `cin, a, b, sum` and `cout` are the arguments of the hardware function, representing the interface of the adder. Inside the hardware function, structures are built from modules and signals. Gates `AND`, `OR` and `XOR` are pre-defined modules; they are composed together by operator "||". Signals with the same name are connected. `aXb, ab` and `abc` are internal signals declared by the `intern` declaration.

```
hw  fullAdder (cin, a, b, sum, cout) =
  let
    intern aXb, ab, abc
  in
      XOR (a, b, aXb)
   || XOR (cin, aXb, sum)
   || AND (cin, aXb, abc)
   || AND (a, b, ab)
   || OR  (ab, abc, cout)
  end
```

Figure 2.2: Structural description of a 1-bit fullAdder

Figure 2.3 gives the behavioral description of the full adder. The behavior description is the direct translation of the formulas for calculating `sum` and `cout`. It consists of two signal assignments (each is a *behavior*) composed together by a behavior operator "||" (same as that for structure).

```
hw  fullAdder (cin, a, b, sum, cout) =
    sum := a xor b xor cin
 || cout := (a and b) or (cin and (a xor b))
```

Figure 2.3: Behavioral description of a 1-bit fullAdder

2.2 HML Types and Objects

The advanced type system is HML's most prominent feature compared to other hardware description languages. HML is strongly typed yet polymorphic. This makes HML functions more flexible and at the same time strong type-checking prevents run-time type errors and some design rule violations. Automatic type inference gives users the convenience of not specifying types. Section 2.2.1 describes the basic types of HML. Section 2.2.2 discusses advanced types built on basic types. Section 2.2.3 explains HML objects and how to declare objects, particularly *signals*.

2.2.1 HML Basic Types

HML provides a number of basic types, including *unit*, *bool, bit, integer* and *bit-vector* types. The operators on these types are defined in Section 2.3.3.

1. *Unit* is the most basic type with a single value written as "()". An expression with a value of unit is evaluated as a *null* expression.

2. *Bit* type has a value of either *high* written as `'1'` or *low* written as `'0'`.

3. *Bool* can have a value of either *true* or *false*, it is mainly used in the software sense in behavioral description, while *bit* has a hardware interpretation.

4. An *integer* type is a range of integer values within the range specified by declaration *range* or the default range is used. The syntax of a *range* declaration is:

```
range   integer1, integer2
```

`integer1` and `integer2` are two arbitrary integers. At the stage of synthesis, integers are synthesized into bit-vectors. The width of the bit-vectors are decided by the range of the integers.

5. A *bit-vector* type is an array of *bit*s. The width of bit-vectors used in a program can be set by a *width* declaration. The syntax is:

```
width   integer1,integer2
```

`integer1` and `integer2` are two non-negative integers that represent how the bit-vector is labeled.

Note that in an HML program, *range* or *width* declaration can be used only once, as the global integer range or bit-vector width. If an object has different range or width from what is declared by the *range* or *width* declaration, it should be specified when the object is introduced for the first time. For example, in the following program segment,

```
width 7,0
hw example (a : bit\_vector(3,0), b, c...) =
     ...
```

`width 7,0` declares that all the bit-vectors used in this program has width of 8-bit (labeled as bit-vector(7 downto 0)), excepted where noted. `a:bit_vector(3,0)` specifies that the width of `a` is 4-bit (labeled as bit-vector(3 downto 0)).

6. HML has a special type called *behavior*. It is used in hardware function declarations as the return type. Signal assignments allow behaviors to communicate with each other; they are also the primitives of behavior types. There will be more detailed discussion about behavior type and signal assignments in this chapter.

2.2.2 Advanced Types

As a high-order language, HML's functions are first-class objects and can be used in other functions as arguments. Functions have *function types* which are mappings from a list of types to basic types. The **definition** of *function type* is as follows.

`function_type := type list` $\rightarrow$ `type`

The type list at the left side of the $\rightarrow$ is a list of the input argument types. The type at the right side of the $\rightarrow$ is the type of the return value of the function.

For example, `fun add (a, b) = a + b` defines a polymorphic `add` function with type : $\alpha * \alpha \rightarrow \alpha$. α is a type variable who value can be either integer type or bit-vector type. Type variable is used in polymorphic type system to represent unknown types. One of its important applications is to check consistent usage of an object whose type is unknown [ASU86].

SML's function definition is extended to include the notion of *hardware function*. A *hardware function* type is the type of a hardware module. It is a mapping from an interface type to a behavior type. The interface type is a list of hardware interface ports, each port is a pair of Input/Output information and type. The Input/Output can have four values: `Input` means the port is an input; `Output` means

that it is an output; `InOut` means that it is a bidirectional port; `Non_applied` implies that an object is used in the software sense only and therefore does not have input/output meaning. A *hardware function* returns a *behavior* type. Here is the definition of hardware types:

```
hardware_type := interface_type → behavior_type
interface_type := (io, type) list
io := Input | Output | InOut | Non_applied
```

In our 1-bit full adder example, the hardware function `fullAdder` has three inputs `a, b` and `cin` and two outputs `sum` and `cout`; its type is:

```
(Input,bit) * (Input,bit) * (Input,bit)
* (Output,bit) * (Output,bit) → behavior.
```

Because HML supports high-order functions, the term `type` used in `function_type` and `interface_type` can be either basic types or advanced types including function types and hardware function types.

An *instance* of an HML module (with a *hardware function* type) is the application of the corresponding hardware function to some signals that it is connected to; it has a behavior type. For example, `XOR (a, b, aXb)` as an instance of hardware function `XOR` is of behavior type.

HML allows **user defined types**. New types can be created with *type* declarations. A `type` declaration defines an abbreviation for an *enumerated type* expression. An enumerated type is an ordered set of identifiers. The identifiers within a single enumerated type must be distinct, they should also be distinct from other identifiers used as variables or signals.

An example of an enumerated type in HML is:

```
type state = add | subtract | multiply | divide .
```

Type `state` is defined with four possible values.

Table 2.1 gives a summary of HML types.

Table 2.1: HML Types

Types		Descriptions
Basic types	Unit	()
	Bool	true, false
	Bit	'0', '1'
	Integer	Range of integers, `range i, j`
	Bit-vector	Array of bits, `width i, j`
	Behavior	Type of signal assignments
Advanced types	Function type	Type of regular functions `function_type = type list → type`

	Hardware function type	Type of hardware modules
		`hardware_fun =`
		`io_type list → behavior_type`

User defined types		Enumerated types `type t = v1 | v2 | ...`

2.2.3 HML Objects and Object Declarations

An object is a named item in an HML description which has a value of a specified type. There are two classes of objects: *constants* and *signals*.

Constants can be declared with a *val* declaration (value declaration) at the global level. Value declarations assign values to an object, they have the syntax :

val *id* $=$ *exp*.

Signals can also be viewed as a type. They are used to communicate between hardware behaviors in a behavioral description or to connect submodules in a structural design. They have type $\alpha^{=}$. The superscript "=" indicates that we require signal values to be of an equality type. If a type can have an equality test as an operation then it is called an equality type. The basic types *int, bool, bit, unit* and *bit-vector* are all equality types. On the other hand, any type involving a function type ($T_1 \rightarrow T_2$) is not an equality type [Rea89].

All the arguments in a hardware function are signals that are visible at a mod-

ule's interface. The signals used internally in a hardware module can be declared by *intern* declarations inside hardware function declarations. The syntax for *intern* declarations is :

intern *id1, id2, ids ...*

In the full adder example in Figure 2.2, declaration `intern aXb, ab, abc` declares `aXb, ab` and `abc` to be internal signals that are invisible outside the module.

Signals can also be declared by a *val* declaration, with its initial value set as the value of the *exp* in the *val* declaration. For example, in the following program segment, `val in_signal = 0` declares an internal signal `in_signal` of type *int* and sets its initial value to be `0`.

```
hw  example(...) =
  let
     val in_signal = 0
  in
     ...
  end
```

Only *val* declarations that appear inside a hardware function declaration declare signals. Global level *val* declarations declare constants.

Due to HML's automatic type inference system, an object declaration does not have to specify the object's type. However, as an option, the user can choose to specify types of objects. The syntax for specifying types is to add type information when an object is first declared. For example, `val in_signal : int = 0` declares signal `in_signal` to have type *int*. This is the type that would be inferred automatically if it were not declared.

2.3 HML Programming Constructs

This section describes the HML programming constructs, including lexical elements, expressions and operators, and declarations. The syntax for these constructs are given. For more details about HML syntax, refer to Appendix A.

2.3.1 HML Lexical Elements

1. Comments

Comments in HML are enclosed by "`(*`" and "`*)`". Nested comments are supported.

2. Identifiers

Identifiers are used as programmer defined names. They must conform to the rule:

```
id :  [A-Za-z][A-Za-z_0-9]*
```

3. Bit and Bit Strings

The value of a bit is represented by `'0'` or `'1'`. Bit-vectors are arrays of type bit, they are represented by enclosing the bit vector value in double quotes, for example `"00011101"`.

4. Bool Value

The value of `Bool` type is either "`true`" or "`false`".

5. Key words

The following names are reserved for key words: **and, andalso, bit, bool, bit-vector, case, else, end, false, fn, fun, hw, if, in, int, intern, inv, let, nand, nor, not, of, or, orelse, structure, then, true, type, val, xor,** and **xnor**.

2.3.2 Signal Assignments

As mentioned in Section 2.2.1, signal assignments are the primitives of behavior type. In HML, signals can be assigned values by two kinds of assignments. Table 2.2 shows their syntax and hardware meanings. The first, combinational

assignment, is intended to model the behavior of combinational logic. The target signal is updated immediately. The second, register assignment, is intended to model the behavior of sequential circuit elements; the target signal is updated in the next clock cycle. Assuming global time `t` with values of 0, 1, 2, ..., the signal assignment syntax can be seen as a shorthand for the hardware meanings shown in Table 2.2. To keep HML simple, the above signal assignments do not specify any timing information. However, register assignments implicitly include clock information. The HML-to-VHDL translator that we have implemented will prompt users about whether to add timing information to the generated VHDL code. Based on the user's choice, timing information can be automatically added to the VHDL code by the translator. Since timing and clocks do not need to be specified, the programs emphasize functionality and ignore the actual implementation details; they are easier to read or write.

HML only supports one global clock; users can choose the type of the clock while compiling the program. The support for multiple clocks is future work.

Table 2.2: Signal assignment: syntax and hardware meanings

Combinational Assignment	Syntax	`signal := expression`
	Meaning	signal(t) = expression(t)
Register Assignment	Syntax	`signal <- expression`
	Meaning	signal(t+1) = expression(t)

2.3.3 Expressions and Operators

An HML expression is a formula combining primitive expressions with operators or with programming constructs such as `if..then..else`. All the HML operators and their precedence and scope are listed in Table 2.3; they are discussed below.

Table 2.3: HML Operators and precedence (in order of decreasing precedence)

Class	Operators	Types operated on & descriptions
Sign	~	int
Arithmetic	* , div	int, bit-vector
	+, -	
Relational	=,<>, >, >=, <, <=	int, bit-vector
Logical	and, or, nand, nor, xor, xnor, inv	int, bit, bit-vector
Boolean	andalso, orelse, not	bool
Bit concatenation	@	bit, bit-vector
Behavior	;	In behavioral description: sequential expressions
	||	Submodule composer and Concurrent behavior composer

2.3.3.1 Operators

The **arithmetic operators** are **+, -, ***, and **div**; they operate on values of type *integer* or *bit-vector*. Bit-vectors are assumed to be 2's complement representations of integers in these operations.

The **boolean operators** are **not**, **andalso** and **orelse**. **andalso** and **orelse** are "short-circuit" operators; they only evaluate their right operand if the left operand does not determine the result. They all operate on *bool* type.

The **logical operators** are **and, or, nand, nor, xor, xnor** and **inv**, they operate on values of type *integer, bit* and *bit-vector*. The operators are bitwise if operating on integers and bit-vectors. Integers are treated as bit-vectors with 2's complement representation.

The **relational operators =,>, >=, <, <=, <>**, are used on values of type *integer* and *bit-vector*. Bit-vectors, are treated as 2' complement representations of integers. = and <> can be also applied to values of type *bit* and *bool* to compare whether the two values are equal.

The **sign operator** $\sim$ operates on values of *integer* type. Note the difference from "$-$".

All operators of two operands require the two operands to have the same type.

2.3.3.2 Bit-vectors Operations

Among the operators described in Section 2.3.3.1, arithmetic, logical and relational operators can be used with bit-vectors. Bit-vectors also have some special operations that are very useful; these includes bit selection and concatenation.

1. Bit selection: The bits in a bit-vector can be selected by a bit selection expression, with the syntax:

`bit-vector-name [ integer ]` , for selecting a single bit and

`bit-vector-name [ integer1, integer2 ]` for selecting a range of bits.

For example, if `BV` is a bit-vector(7,0) of value `"00011101"`, then `BV[0]` selects the least significant bit, which is `'1'`; `BV[7,4]` selects the most significant four bits, which are `"0001"`.

2. Concatenation: Two bit-vectors can be concatenated into a new bit-vector by operator "@". The new bit-vector's width is the sum of the two primitive bit-vectors; it is labeled as bit-vector(width-1,0) regardless of how the two primitive bit-vectors are labeled. For example, if `b1(3,0)` is "0111" and `b2(3,0)` is "1010", then `b1 @ b2` is "01111010" and is labeled as (7,0). If `b1` and `b2` are labeled differently but keep the same values, `b1 @ b2` is unchanged.

Bit-vector concatenation can not only be used in expressions, such as in `b3 := b1 @ b2`, but can also be used for pattern matching in the left-hard side of signal assignments. The example in Figure 2.4 uses bit-vector concatenation and describes an adder with carry output. Inputs `a` and `b` are bit-vectors of the same width (assume the width is `w1`). `sum` is the sum result of width `w1` and `carry` is the 1-bit carry output.

```
width ...
hw adder (a, b, sum, carry:bit) =
carry @ sum := a + b
```

Figure 2.4: HML description of an adder with carry output using bit-vector concatenation

Concatenation is not supported by the HML system yet.

2.3.3.3 Behavioral Operators

There are a couple of operators that operate on *behavior* type. Behavior operator "||" is used to compose several submodules into one hardware module – in structural descriptions, the submodules are instance of hardware functions and have type of *behavior*; in behavioral descriptions, the submodules are concurrently running behaviors. The submodules (in either structural format or behavioral format) communicate via signals. Structural and behavioral descriptions can be mixed by "||".

Operator ";" is used inside a behavioral description to compose multiple behavioral expressions into a behavior. ";" is also used in software functions, as the sequential expressions composer; in this case, ";" operates on any type except *behavior*. The use of ";" in software functions will be discussed in Section 2.3.3.4.

Behavioral operators "||" and ";" have the same hardware meanings. For example,

`hw ...  = s:=1; t:=2` and

`hw ..  = s:=1 || t:=2`

are equivalent. The difference is syntax: "||" is only allowed in the top-level of a hardware function to compose either behaviors or structural modules; the HML parser treats behaviors composed with "||" as independent processes. ";" is usually used inside a behavior at a lower level when the behavior has multiple expressions

in it. For example,

```
hw ...= if condition then (s:=1; t:=2) else ...
```

is valid in HML; the ";" used here is not replaceable by "||" because it appears inside an *if-then-else* expression but not at the top-level. Users can make use of the syntax difference to organize their programs: use "||" to partition the whole circuit into several sub-modules; use ";" while describing each sub-module.

While using "||" and ";" to compose behavior expressions, the order is not important because all the behavior expressions execute concurrently.

2.3.3.4 Expressions

In addition to the operators described above, HML has following constructs for expressions. Key words are **bold-faced**.

- *If-then-else* expression : **if** *exp1* **then** *exp2* [**else** *exp3*].
  The *else* part is optional.

- *Let-in-end* expression : **let** *decls* **in** *exp* **end**.
  decls is multiple declarations.

- *Case* expression :

  case *exp* **of** *rule list*,

  rule : *pat* => *exp*.

If **multiple expressions** are use in any place where one *exp* appears, they must be enclosed by " (" and ") ", and they are separated by ";". In software functions, these expressions can have different types and they are evaluated in order; the result for the whole expression is that of the last expression in the sequence. In hardware functions, these expressions must have the type of *behavior*.

Signal assignments are expressions which are of behavior type. This has been discussed in Section 2.3.2.

Functions in HML are abstract values; They need not have a name. HML allows **anonymous functions** with *fn* notation as shorthand for function declarations. *fn* is an expression which has function type, it defines an anonymous software function. The syntax is:

Fn expression : **fn** *rule list*. The definition of *rule list* is the same as defined in *Case* expression.

For example, `fn n=> n * 2` is a function of type $int \rightarrow int$ that doubles an integer. It is an anonymous form for function defined by:

`fun double(n) = n * 2.`

Fn expressions can be applied to an argument; `(fn n=> n * 2)(9)` applies the function defined by `(fn n=> n*2` to an integer argument 9. Anonymous functions can be given a name by a `val` declaration. `val double = fn n=> n * 2` is equivalent to `fun double(n) = n * 2`.

Note that *fn* notation only defines software functions.

2.3.4 Function and Hardware Function Declarations

There are two kinds of functions in HML, the software *functions* and *hardware functions*, as discussed in Section 2.2.2. The types of them have been addressed in Section 2.2.2. Here we look at their syntax and difference.

The syntax for function and hardware function declarations are very similar except the two have different keywords **fun** and **hw**. The syntax is:

fun_hw_dec : **fun/hw** *clauses*

clauses : *clause*

| *clause* | *clauses*

clause : **id** pats [: *result_type*] = *exp*.

Note that in software or hardware function declarations, the types of arguments and results do not need to be declared. The user can choose to declare all of them

or some of them; this is useful particularly when some types can not be resolved by the type checker. For example,

```
fun sum (a, b) = a+b
```

defines a polymorphic adder that can operates on *integer* or *bit-vector* types. A user who wants an integer adder can write:

```
fun sum (a:int, b) = a+b
```

which declares input `a` to have integer type, or

```
fun sum (a, b):int = a+b
```

which declares the return value to have integer type.

Semantically, the major difference between the two kind of declarations is that hardware functions represent hardware modules, while software functions are for software use. They can be called in hardware functions, and are used only as an aid to hardware functions. Another difference is that a software function returns values of a specific type other than behavior type, and a hardware function must return behavior type (which means the expression in the hardware function declaration must be composed by signal assignments or other hardware modules). Hardware function declarations are the major parts of an HML program; how to declare hardware functions will be covered in greater detail in Section 2.4.

2.4 Describing Hardware in HML

HML is a declarative language. Figure 2.5 shows that the top level of HML programs are a series of declarations, including *value* declarations, *type* declarations, *function* declarations and *hardware* declarations. *Hardware* declarations introduce a special kind of function which represents hardware modules; they are the main part of HML descriptions.

HML *hardware functions* can be used to declared structures or behaviors or the mixture of both structures and behaviors.

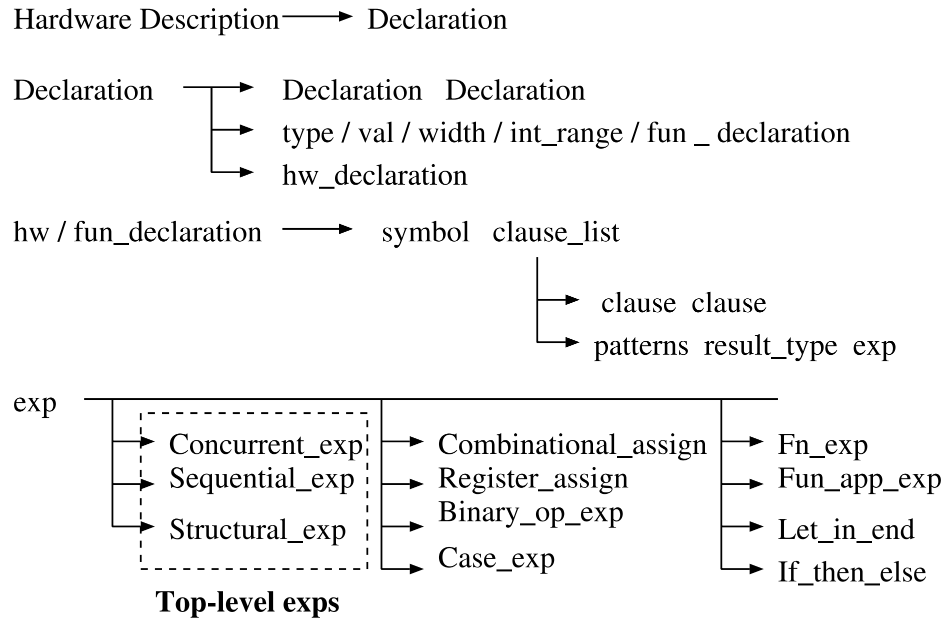

Figure 2.5: HML programming constructs and program organization – a simplified HML abstract syntax tree

2.4.1 Describing Structure in HML

HML structures are built from modules and signals. Modules may either be primitive modules or compound modules built using the module composition operator "||". Primitive modules are predefined hardware functions described with structural or behavioral HML. HML provides a small predefined library of basic gates that can be used to build other structures. Signals with the same name are connected. Signals internal to a module are declared using the `intern` keyword and scoped by `let...in...end`. The arguments of the modules are signals.

In Section 2.1.3, we have seen a 1-bit full adder described in structural HML: given the two-input gates `AND`, `OR` and `XOR` have been defined, a one bit full adder is defined as an interconnection of these primitives. This kind of structural description is straightforward. It is like a language version of the schematic: list all the submodules and draw the connections. Other languages that support structural level descriptions such as VHDL also use similar formats, although the actual

syntax may vary. However, since a hardware module can be an argument of another hardware function, HML structural descriptions can be more complicated and more flexible than a simple netlist.

Using the structural format, functions for regular structure generators can be written in HML by exploiting polymorphism and high order functions. These can then be applied to different types of cells. We have written generators for different kinds of arrays and trees. In VHDL and other HDLs that do not support polymorphism and high order functions, the user has to write the same generators several times if the type of the cell structures or the number of the cells is different.

Example. The array generator is independent of the actual types of its cells. A hardware function `array` instantiates `n` copies of `cell` connected to buses `inp` and `outp`. See Figure 2.6. This array generator is polymorphic: `cell` can be of any module with an input and an output port that can be of any width; `inp` and `outp` are bit-vectors of any width.

```
hw  array 1 cell inp outp =
      cell (inp[0], outp[0])
  | array n cell inp outp =
        cell (inp[n-1], outp[n-1])
     || array (n-1) cell (inp, outp))
```

Figure 2.6: Example: A polymorphic array generator

Example. Based on the 1-bit full adder described in Section 2.1.3, we can write an n-bit full adder generator (see Figure 2.7). If the 1-bit full adder is generalized to other cells with the same interface, then the following description becomes a generator of a ripple array with outputs.

Note that for high-order and recursive functions, we only implement type checking but not the translation to VHDL. This will be discussed in detail in Section 4.3.

```
hw adder-array(1, fullAdder, a, b, cin, sum, cout) =
        fullAdder (a[0], b[0], cin, sum[0], cout)
 | adder-array(n, fullAdder, a, b, cin, sum, cout) =
     let
        intern carry
     in
         fullAdder  (a[n-1], b[n-1], carry, sum[n-1], cout)
      || adder-array(n-1, fullAdder, a, b, cin, sum, carry)
     end
```

Figure 2.7: An n-bit full adder generator based on a 1-bit full adder

2.4.2 Describing Behavior in HML

A behavior is a group of hardware description expressions based on signal assignments. Inside a behavior, the hardware description expressions are evaluated *sequentially* in zero time in each evaluation cycle. HML has a behavior composer "||" that combines several *concurrently* executing behaviors into a hardware function of a behavior type.

Example. The simple adder shown in Figure 2.8 can operate on various types (*integer, bit or bit-vector*). The assignment `s1 := a + b` specifies a combinational adder and `s2 <- s1` specifies that the output `s1` will be latched in a register.

```
hw sum (a, b, s1, s2) =
       s1 := a + b
   ||  s2 <- s1
```

Figure 2.8: Example: A polymorphic adder with latched output

The program doesn't specify a clock, but the "`<-`" signifies a register assignment which implies that a clock signal is needed in the hardware. Also no variable or signal declarations are needed and the arguments of the top level function represent the interface (inputs and outputs) to the hardware described.

2.4.3 Putting Behaviors and Structures Together

In the previous sections we have discusses how to describe behaviors and structures in HML. Sometimes it is convenient to mix behavioral descriptions and structural descriptions. Mixing behavioral and structural descriptions is particularly useful when users try to build new programs based on some predefined hardware functions (modules) – the predefined hardware functions can be used directly in a structural format while new descriptions are added in either structural or behavioral format.

Assuming we have predefined an `add` hardware function,

```
hw add (x, y, z) = z:=x+y,
```

the polymorphic adder with latched output in Figure 2.8 can be rewritten by mixing structural and behavioral descriptions, as shown in Figure 2.9.

```
hw sum (a, b, s1, s2) =
        add (a, b, s1) (*structural format*)
    ||  s2 <- s1       (*behavioral format*)
```

Figure 2.9: Rewrite the polymorphic adder with latched output by mixing structural and behavioral descriptions

Figure 2.5 summarizes the basic programming facilities and program organization of HML. It is a simplified HML abstract syntax tree. For greater detail on the syntax of HML, refer to Appendix A.

2.4.4 Restrictions of Describing Hardware in HML

There are some restrictions that users should follow in order to write correct HML hardware descriptions.

No combinational loops are allowed: combinational loops do not have specific hardware meanings. For example, if there exist two combinational signal assignments `s:=t` and `t:=s`, the type checker will detect this as an error. However, `s:=t` and `t←s` can appear in the same program because the register assignment `t←s`

introduces a delay (a clock cycle). In real programs, combinational loops may have more complicated formats; the detection of such loops is done by detecting whether there is any cycle in the signal dependency graph for combinational signal assignments only.

Multiple assignments to the same signal are illegal in HML. Since all the signal assignments run concurrently, multiple assignments to the same signal can not be resolved and therefore are not allowed. The idea for implementing this restriction is simple – to detect whether the left-hand side signal in a signal assignment appears in the left-hand side of other signal assignments. In the case of mixing structural and behavioral descriptions, it is a little more complicated. Consider the example in Figure 2.10; `s1` appears to be assigned only once, but it is also the output of `AND(a, b, s1)`. In this case `s1` actually gets assigned twice. Therefore the strategy to detect multiple assignments is generalized to detect not only whether a signal appears at the left-hand side of an assignment but also whether it is an output of a structural expression.

```
hw example (a, b, c, s1, s2) =
        AND (a, b, s1)
    || s1 := a + b
    || s2 := a OR c
```

Figure 2.10: Example of multiple assignments to one signal

2.5 Discrepancies Between HML and SML

As discussed in previous sections of this Chapter, HML inherited a big part of its syntax from SML. However, HML differs somewhat from the syntax inherited from SML. Users who have used SML before can avoid confusion by following these guidelines. Users who do not have SML experience can ignore this section.

- **Functions:** SML allows two formats for defining functions of multiple arguments. For example, for an `Add1` function that adds two values, both

  `fun Add2 x y = x + y`

  and `fun Add (x, y) = x + y`

  are acceptable. The two `Add` functions defined have different types: the former format has type $\alpha \rightarrow (\alpha \rightarrow \alpha)$; the latter has type $\alpha * \alpha \rightarrow \alpha$. The former format defines a *curried function* [Pau91] and permits *partial application*. In our example, if `Add1` is applied to its first argument (of type α) its result is a function of type $\alpha \rightarrow \alpha$. As in SML, HML also allows these two formats of defining functions of multiple arguments. HML however does not support the feature of *curried functions* and the two formats define functions of the same type: $\alpha * \alpha \rightarrow \alpha$. *Partial application* is not supported in HML.

- **Keyword AND:** In HML the declaration construct `val... and val ...` is not permitted and keyword **and** is reserved as a logical operator.

- **Let-in-end:** In the `let-decl-in-exps-end` construct, the multiple expressions `exps` doesn't need to be put in parentheses; but in HML, they have to be put in parentheses. For example, `let .. in exp1; exp2; exp2 end` is valid in SML, but in HML, only `let .. in (exp1; exp2; exp2) end` is permitted.

Chapter 3

HML Type Checking and Inference

This chapter examines type checking and type inference in HML. The type checking rules and the algorithm are discussed.

3.1 Introduction

Since HML is a polymorphic, high-order programming language, type checking and type inference are more important and complicated than for a language with a monomorphic type system and no high-order functions.

The task of the type checker is to check that an expression has a particular type or to infer a most general type for it if its type can not be decided. A type inference algorithm can be constructed for deducing the types of expressions from the types of primitive operations [Rea89].

One of the principal goals of HML is to catch errors as early in the design process as possible. This is done by performing some design rule checking as part of type checking. The design rule checking done by the HML type checker is mostly interface checking: if a signal or a bus connected to a hardware module does not

match its port type, then an error occurs.

The type inference information is very crucial for HML to connect to any back end. In the HML-to-VHDL translator that will be described in the next chapter, type information is added to all VHDL declarations.

3.2 Type Checking and Type Inference Rules

3.2.1 Definition of Types Used in the Type Checker

The definition of types (see Figure 3.1) in the type checker not only includes the general meaning of types, which is called *pure types* here, but also includes *modes* in types, so that we can tell whether an expression has a **constant** mode or a **variable** mode. The difference is that an expression of *constant* mode can not be assigned to. The following classes of type metavariables are needed to show the type checking and type inference rules of HML:

1. **Modes** μ **:** Ranges over modes; either **constant** or **variable**.

2. **Pure types** π **:** Ranges over types without modes attached, for example, **int** or **bool**.

3. **Types** τ **:** A type is a mode plus a pure type; that is, every τ can be expressed in the form $\mu\pi$, such as **constant int**.

4. **In/Out** ι **:** This is used in *hardware* types only, to indicate whether a port of the hardware is input, output, or a bidirectional port.

Because of the polymorphism of the type system, the possible forms of type are extended to include type variables α, β, γ, ...

$$
\begin{array}{lll}
\mu & := & \textbf{constant} \mid \textbf{variable} \\
\pi & := & \textbf{unit} \mid \textbf{bool} \mid \textbf{int} \mid \textbf{bit} \mid \textbf{behavior} \\
 & \mid & \textbf{bit_vector}\ (i : \textbf{int},\ j : \textbf{int}) \\
 & \mid & \textbf{function}\ (\pi_1, ..., \pi_n) \rightarrow \pi \quad \text{where n} \geq 0 \\
 & \mid & \textbf{hardware}\ (\ \iota_1\pi_1, ..., \iota_n\pi_n) \rightarrow \textbf{behavior} \quad \text{where n>0} \\
\iota & := & \textbf{input} \mid \textbf{output} \mid \textbf{inout} \mid \textbf{not_applied} \\
\tau & := & \mu\pi
\end{array}
$$

Figure 3.1: Definition of types: modes and pure types

3.2.2 Environments

We need to keep track of the types of all the objects; they are included in type *environments*, which are simply partial functions from objects to types. While describing type checking rules, **A** is used to represent environments.

3.2.3 Type Inference Rules

The rules for the polymorphic type inference system are summarized in Figure 3.2 [Rea89]. This section explains all the type inference rules individually.

Before explaining the inference rules in Figure 3.2, we look at the notation used to describe the rules. In expression "A $\vdash E : \mu\pi$", "A" is the environment discussed in Section 3.2.2, E is an expression, and $\mu\pi$ is the mode and pure type. "A $\vdash E : \mu\pi$" means that in environment A, expression E has mode μ and type π.

1.Signal	A $\vdash$ V : S	when A(V) = S
2.Constant	A $\vdash$ C : S	when A(C) = S

Rule 1 and 2 indicate that to check what type an identifier has, look it up in the environment. In HML, an identifier can be either a signal or a constant.

3.Arithmetic, logical & boolean exps
$$\frac{\text{A} \vdash E_1 : \mu_1\pi \qquad \text{A} \vdash E_2 : \mu_2\pi}{\text{A} \vdash E_1 \star E_2 : \textbf{constant}\ \pi}$$

0. Notation	$A \vdash E : \mu\pi$
1. Signal	$A \vdash V : S$ when $A(V) = S$
2. Constant	$A \vdash C : S$ when $A(C) = S$
3. Arithmetic, logical & boolean exps	$\dfrac{A \vdash E_1 : \mu_1\pi \qquad A \vdash E_2 : \mu_2\pi}{A \vdash E_1 \star E_2 : \textbf{constant}\ \pi}$
4. Relational expressions	$\dfrac{A \vdash E_1 : \mu_1\pi \qquad A \vdash E_2 : \mu_2\pi}{A \vdash E_1 \star E_2 : \textbf{constant bool}}$
5. Signal assignments	$\dfrac{A \vdash E_1 : \textbf{variable}\ \pi \qquad A \vdash E_2 : \mu\pi}{A \vdash E_1 := / \leftarrow E_2 : \textbf{constant behavior}}$
6. If-then-else	$\dfrac{A \vdash E_1 : \mu_1\ \textbf{bool} \quad A \vdash E_2 : \mu_2\pi \quad A \vdash E_3 : \mu_3\pi}{A \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \textbf{constant}\ \pi}$
7. Case expression	$\dfrac{A \vdash E^{'} : \mu^{'}\pi^{'} \quad A + [\ V : \textbf{constant}\ \pi^{'}\ ] \vdash E : \mu\pi}{A \vdash \text{case } E^{'} \text{ of } V \Rightarrow E : \textbf{constant}\ \pi}$
8. Function application	$\dfrac{A \vdash E : \mu_1\ \ \pi^{'} \rightarrow \pi \qquad A \vdash E^{'} : \mu_2\pi^{'}}{A \vdash EE^{'} : \textbf{constant}\ \pi}$
9. Let-in-end	$\dfrac{A \vdash E^{'} : S \qquad A + [\ V : S\ ] \vdash E : \mu\pi}{A \vdash \text{let val } V = E^{'} \text{ in } E \text{ end} : \textbf{constant}\ \pi}$
10. Function abstraction	$\dfrac{A + [\ V : \textbf{constant}\ \pi^{'}\ ] \vdash E : \mu\pi}{A \vdash \text{fn } V \Rightarrow E : \textbf{constant}\ \ \pi^{'} \rightarrow \pi}$

Figure 3.2: HML type checking and type inference rules

The types of composite expressions are built from the types of their subterms. **Rule 3** explains the type inference for arithmetic, logical and boolean expressions. Let $\star$ be any operator of the above classes. The rule says that the two sub-expressions can have any mode $\mu 1$ and $\mu 2$, but they must have the same type π; the result expression has **constant** mode (for example, **x+y** has constant mode – x+y := 1 is not a valid expression), and type π - the type of the two sub-expressions.

4.Relational expressions $$\frac{\mathrm{A} \vdash E_1 : \mu_1 \pi \qquad \mathrm{A} \vdash E_2 : \mu_2 \pi}{\mathrm{A} \vdash E_1 \star E_2 : \textbf{constant bool}}$$

Type inference rule for relational expressions (**rule 4**) is similar to rule 3 – it requires that the two sub-expressions to have the same type but not necessarily the same mode. The result expression has constant mode and *boolean* type.

5.Signal assignments $$\frac{\mathrm{A} \vdash E_1 : \textbf{variable}\ \pi \qquad \mathrm{A} \vdash E_2 : \mu \pi}{\mathrm{A} \vdash E_1 := / \leftarrow E_2 : \textbf{constant behavior}}$$

For signal assignments (either combinational or sequential), as in **rule 5**, the left-hand side and right-hand side sub-expressions should have matching types; the mode of the left-hand side expression must be **constant** mode. The result expression is of *behavior* type, as discussed in Section 2.3.2.

6.If-then-else $$\frac{\mathrm{A} \vdash E_1 : \mu_1\ \textbf{bool}\ \mathrm{A} \vdash E_2 : \mu_2 \pi\ \mathrm{A} \vdash E_3 : \mu_3 \pi}{\mathrm{A} \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \textbf{constant}\ \pi}$$

If-then-else expressions also require matching types; the type of the condition must be *boolean* (**rule 6**).

7.Case expression $$\frac{\mathrm{A} \vdash E' : \mu' \pi'\ \mathrm{A} + [\ \mathrm{V} : \textbf{constant}\ \pi'\ ] \vdash E : \mu \pi}{\mathrm{A} \vdash \text{case } E' \text{ of V} \Rightarrow E : \textbf{constant}\ \pi}$$

In a case expression (**rule 7**) case $E^{'}$ of V $\Rightarrow E$, expression $E^{'}$ and its case V should have the same type; the result expression type is the type of case body expression E. The real case expression usually has multiple clauses; cases and body expressions in different clauses should be compared to each other to guarantee matching types. Type information of one clause can also be used to infer types of another clause.

8.Function application $\dfrac{\text{A} \vdash E : \mu_1 \;\; \pi^{'} \rightarrow \pi \;\; \text{A} \vdash E^{'} : \mu_2 \pi^{'}}{\text{A} \vdash EE^{'} : \textbf{constant}\ \pi}$

A function expression has type $\pi^{'} \rightarrow \pi$ where $\pi^{'}$ is a list of the types of the arguments and π is the return type. In a function application (**rule 8**), assuming the type of the function is $\pi^{'} \rightarrow \pi$, the arguments are checked to have type $\pi^{'}$; the result expression has type π and constant mode.

9.Let-in-end $\dfrac{\text{A} \vdash E^{'} : \text{S} \qquad \text{A} + [\ \text{V} : \text{S}\ ] \vdash E : \mu\pi}{\text{A} \vdash \text{let val V} = E^{'} \text{ in } E \text{ end} : \textbf{constant}\ \pi}$

The type inference rule for let-in-end expressions (**rule 9**) is more complicated because the declarations in a `let-decls-in-exp-end` expression introduce new bindings and therefore the body expression `exp` should be evaluated in a new environment. The result expression type is the type of the body expression.

10.Function abstraction $\dfrac{\text{A} + [\ \text{V} : \textbf{constant}\ \pi^{'}\ ] \vdash E : \mu\pi}{\text{A} \vdash \text{fn V} \Rightarrow E : \textbf{constant} \quad \pi^{'} \rightarrow \pi}$

Function abstraction (fn-expression) is similar to the let-in-end expression because it also involves new bindings. In expression fn V $\Rightarrow E$, function argument V should be added into the old environment A while evaluating body expression E. If the type of V is inferred as $\pi^{'}$ and the type of E is π, the result expression has

function type $\pi' \rightarrow \pi$. Like case expressions, function abstractions may have multiple clauses; these clauses should be compared each other to guarantee matching types.

3.3 Type Checking and Inference Algorithm

The basic approach of the type checking and inference algorithm involves two processes: **bottom-up** and **top-down**.

The type checking and inference starts from the leaves of the abstract syntax tree, applies the type rules in Figure 3.2 to check whether the types of the primitives fit the requirement of the rules and to infer unknown types – including types of the parent and the siblings in the abstract syntax tree. For example, in expression `e1+e2`, if subexpression `e1` is checked to have type *integer*, then we can infer that `e2` (sibling) as well the composite expression `e1+e2` (parent) have the same type *integer*.

If a type can not be decided in the bottom-up process, it will be assigned a type variable as its type. If the type of an element in the abstract syntax tree is inferred by checking a type rule, the top-down process is applied to recheck the types of elements that are under the current element in the abstract syntax tree as well as its siblings. Associated with each element is a boolean attribute called `decided` which indicates whether the types of the elements under this elements have been decided. If `decided` is true, there is no need to initiate the top-down process. Top-down processes use the same rules as bottom-up processes, except conditions and results of the rules are reversed.

The processes can be illustrated with the following example.

```
fun type_example (a,b,c,d) = (a+b) + (c+d) + 1
```

The abstract syntax tree segment for expression `(a+b) + (c+d) + 1` is shown

in Figure 3.3.

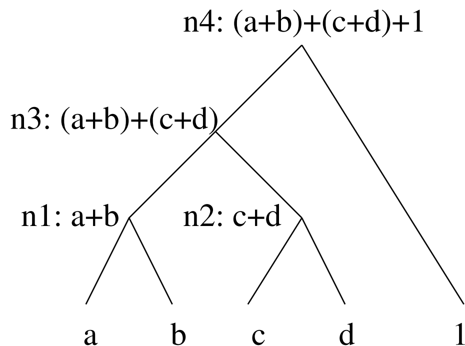

Figure 3.3: Abstract syntax tree segment for expression (a+b)+(c+d)+1

The type checker has to infer the types of arguments `a, b, c` and `d` as well as the type of function `type_example`. When names `a, b, c` and `d` are first introduced as the arguments of function `type_example`, their types are unknown and they are given type variables $\alpha 1$, $\alpha 2$, $\alpha 3$ and $\alpha 4$ as their type expressions. The integer `1` has type *int*. The type inference starts from the bottom-up process. Node `n1:a+b` is checked first according to type inference rule 3 in Figure 3.2. Since `a` and `b` should have same type, the type expression of `b` ($\alpha 2$) is replaced by that of `a` ($\alpha 1$) and the type of node `n1` is $\alpha 1$. Similarly node `n2:c+d` is checked and `c` and `d` as well as node `n2` have type $\alpha 3$. Going up from node `n1` and `n2`, node `n3:n1+n2` can be type-checked. Since `n1:(a+b)` and `n2:(c+d)` should the same type, the type of `n2:(c+d)` $\alpha 3$ is therefore replaced by the type of `n1:(a+b)` $\alpha 1$. This initiates a *top-down* process to update the types of `c` and `d` according to the updated type of `n2:c+d`. In the *top-down* process, `c` and `d` are updated to have type expression $\alpha 1$. Node `n3` has type $\alpha 1$. Then the bottom-up process restarts to check node `n4:n3+1`. The integer `1` force node `n4` as well as node `n3` to have integer type. The updating of `n3` initiates another *top-down* process to update the types of nodes under `n3`; `a, b, c` and `d` are updated to have integer type. The above type inference procedures are summarized with the following steps:

Bottom-up Steps

0. Leaves a:α1, b:α2, c:α3, d:α4, 1:int

1. n1 a+b:α1, update b:α1

2. n2 c+d:α3, update d:α3

3. n3 n1+n2:α1, **top-down** n2:α1, c:α1, d:α1

4. n4 n3+1:int, **top-down** n3:int, n1:int, n2:int, a:int, b:int, c:int, d:int

Based on the types of `a, b, c, d` and type of expression `(a+b)+(c+d)+1` (node `n4`), function `type_example` has type:

int * int * int * int $\rightarrow$ int.

In the type inference algorithm, the bottom-up process is dominant. The top-level type inference function is a bottom-up process based on type inference rules (Figure 3.2). Each bottom-up step may initiate a top-down process if there is updating on any sub-expression. The algorithm is guaranteed to terminate because both bottom-up and top-down processes are one-way recursive traversals of the abstract syntax tree and do not involve any loops. The run time of both processes at a certain node is linear to the depth of the node in the abstract syntax tree.

Chapter 4

Translating HML to VHDL

We have implemented an HML-to-VHDL translator in order to make use of the many VHDL simulation and synthesis tools available. The translation targets a synthesizable subset of VHDL. This chapter first introduces VHDL and its tools, then focuses on HML-to-VHDL translation rules and some of the implementation issues.

4.1 Introduction to VHDL

This section gives a brief introduction to VHDL, describes the VHDL subset that is used in the HML-to-VHDL system and introduces the Mentor-Graphics VHDL simulation/synthesis tools.

4.1.1 VHDL Overview

The development of VHDL, the **VHSIC Hardware Description Language**, was originated in 1980s. In 1987, VHDL was adopted by the IEEE as a standard hardware description language and has since achieved wide spread industrial acceptance [Ash90].

VHDL is a hardware description language with strong emphasis on concurrency.

The language supports hierarchical description of hardware from system to gate or even switch level. VHDL has support at all levels for timing specification and violation detection. It provides constructs for generic design specification and configuration. In addition to the description capability, systems modeled in VHDL can also be simulated at any of the levels in order to verify their functionality.

A VHDL design entity is defined by an *entity declaration* and an associated *architecture body.* The entity declaration specifies its interface and is used by architecture bodies of design entities at higher levels of hierarchy. The architecture body describes the operation of a design entity by specifying its interconnection with other design entities (structure), by its behavior, or by a mixtures of both. The VHDL language groups subprograms or design entities by use of *packages.* For customizing generic descriptions of design entities, *configurations* are used. VHDL also supports libraries and contains constructs for accessing packages, design entities, or configurations from various libraries [Nav93].

4.1.2 The VHDL Subset Used in the HML-to-VHDL System

VHDL was not originally designed for synthesis. Many VHDL constructs are not synthesizable: access types, assert statements, or files for instance have no direct hardware correspondence [OW94]. The translation of HML to VHDL targets a synthesizable subset of VHDL. The VHDL subset used in the HML-to-VHDL system is summarized as four classes in Table 4.1: programming constructs, types, operators, and signal attributes. Note that this is actually a subset of the VHDL synthesizable subset, but it is sufficient to represent all the HML features after the translation.

Table 4.1: The VHDL subset that is used in HML-to-VHDL system

Programming Constructs		Types
Entity declaration	Type declaration	Bit
Architecture body	Wait statement	Bit_vector
Package declaration	Signal declaration	Boolean
Package body	Signal assignment	Enumerated
Process statement	If statement	Integer
Function	Case statement	
Function call	Use clause	

Operators		Signal Attributes
Multiplying	/, *, sla, sra	
Sign	−	'event
Adding	+,−	
Relational	=,/=,<,<=,>,>=	'last_value
Logical	and, or, nand, nor, xor, xnor, not	

4.1.3 Introduction to Mentor-Graphics VHDL Simulation/Synthesis Tools

VHDL-based tools are widely available on platforms ranging from personal computers to multi-user Unix machines. This is one of out major motivations for building the HML-to-VHDL translator. Simulators for full VHDL IEEE 1076 [IEE88] are also available for a variety of platforms. In addition, there are several synthesis programs that take a subset of VHDL as input and generate net lists.

Mentor-Graphics provides a series of commercial VHDL tools including the fully integrated compiler/simulator/debugger design development system *System-1076*, *AutoLogic* VHDL synthesis [Men94a], and *QuickVHDL* simulator [Men94b]. While building the HML-to-VHDL system, we mainly used *QuickVHDL* and *AutoLogic* as simulator and synthesizer respectively.

QuickVHDL is a full IEEE 1076 simulation environment using *Direct-Compiled Code* technology. It allows users to quickly model and test a system at a high level of abstraction. The tool has a graphical interface for inputing VHDL code

and direct viewing of simulation results. It also supports batch operation which is faster and prints output in text format.

AutoLogic synthesizes generic, gate-level implementations from VHDL language models and can optimize a design for area and performance. It supports a subset of VHDL IEEE.

QuickVHDL is integrated with *Design Architect* for schematic or VHDL code entry and *AutoLogic* VHDL for synthesis.

4.2 HML-to-VHDL Translation Rules

HML-to-VHDL translation targets the synthesizable subset of VHDL that is listed in Table 4.1. To appropriately translate HML into VHDL, for each programming construct and object in HML, we need to find its counterpart in VHDL with the same hardware meaning.

This section examines the HML-to-VHDL translation rules, and other implementation issues of the translator. It starts from the top-level translation and then goes further into the lower levels, which include behaviors, expressions, etc. Table 4.2 summarizes the basic translation rules.

4.2.1 Top-level Translation

As discussed in Chapter 2, the core part of an HML program is a hardware function declaration at the top level, which represents a hardware module. In VHDL, this is translated into a VHDL *entity declaration* and *architecture body*, in the main VHDL file. Other top-level declarations (*val, type,* and *fun*) of HML are grouped into a VHDL package, including package declaration and package body. The VHDL package is stored in a separate file. This package is used (with a `USE` clause) in the main VHDL file. The top-level translations are shown in Figure 4.1.

In an HML hardware function declaration, the arguments represent hardware

Table 4.2: HML-to-VHDL translation rules

HML constructs	**VHDL constructs**
Hardware declaration	Entity and architecture
Structure `(f1 || f2)`	Structural architecture, port map
Behavior expression	Behavioral architecture body
Behavior	Process, with sensitivity list
`b1` || `b2`	Concurrent processes inside architecture
`e1 ; e2`	Multiple statements inside process
Global *type/val/fun* declaration	Type/constant/function declaration in package
Local *type/val/fun* declaration	Type/signal/function declaration inside architecture/process
Intern declaration	Signal declaration inside architecture/process
`s := v`	`s <= v`
`s <- v`	`s <= v`, add clock information in the program
Let_decl_in_exp_end	Declaration in architecture/process and statement in process
Fn expression	Function declaration in package/architecture/process and function application
*, div	* , / for simulation sla , sra for synthesis(power of 2)
Object name	Rename, append unique number to the name
`if_then_else` *Case* exp bin_op_exp	Direct translation with syntax change

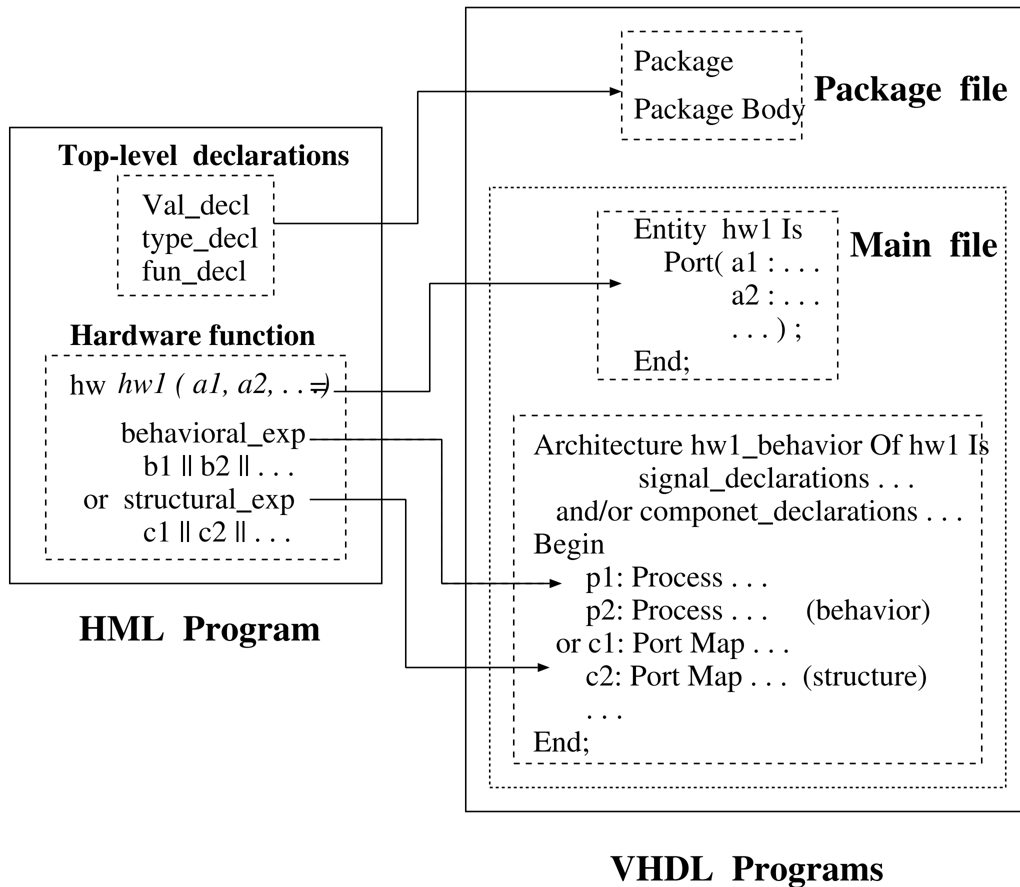

Figure 4.1: HML-to-VHDL Top-level Translation Graph

ports (inputs and outputs). In a VHDL entity, the inputs and outputs have to be specified. Therefore, the HML type checker not only checks the types of the function arguments, but also checks their input and output information. This is done by checking whether the argument is only used on the right-hand side of expressions (implying it should be `IN` in VHDL) or only on the left-hand side of an assignment (`OUT`) or both (`INOUT`).

A valid HML hardware function with the construct `f1` || `f2` ||`...` in which `f1, f2, ...` are instances of hardware functions represents the composition of modules; it is translated into structural VHDL. Otherwise, the HML hardware function has a behavioral form and is translated into a behavioral VHDL description. In Section 2.1.3, a 1-bit full adder example is described using both structural and behavioral formats of HML. Figure 4.2 and Figure 4.3 shows the corresponding VHDL code, which was automatically generated by the HML-to-VHDL translator.

The structural HML full adder is translated into structural VHDL: three components `XOR, AND` and `OR` are declared in the declaration part of the architecture and instantiated inside the architecture; the ports of the instances are mapped into signals and ports of the entity (Figure 4.2).

The behavioral HML full adder is translated into behavioral VHDL with processes (Figure 4.3).

The HML description that contains both structural and behavioral expressions is translated into the hybrid VHDL – structural expressions are translated into component port maps; behavioral expressions are translated into processes. The example used in Figure 2.9 is automatically translated into the VHDL program shown in Figure 4.4.

```
hw  fullAdder (cin, a, b, sum, cout) =
  let  intern aXb, ab, abc
  in  structure( XOR (a, b, aXb);
                 XOR (cin, aXb, sum);
                 AND (cin, aXb, abc);
                 AND (a, b, ab);
                 OR  (ab, abc, cout) )
  end
```

(a) HML structural description of a 1-bit full-adder

```
ENTITY fullAdder IS
   PORT( cin  : IN   bit;
         a    :  IN   bit;
         b    : IN   bit;
         sum  : OUT  bit;
         cout : OUT  bit);
END fullAdder;

ARCHITECTURE fullAdder_structure OF fullAdder IS
   COMPONENT XOR
       PORT(a: IN bit; b: IN bit; c: OUT bit);
   END COMPONENT;

   COMPONENT AND
       PORT(a: IN bit; b: IN bit; c: OUT bit);
   END COMPONENT;

   COMPONENT OR
       PORT(a: IN bit; b: IN bit; c: OUT bit);
   END COMPONENT;

   SIGNAL aXb: bit;
   SIGNAL ab:  bit;
   SIGNAL abc: bit;
BEGIN
   c1: XOR PORT MAP (a => a,   b => b,   c => aXb);
   c2: XOR PORT MAP (a => cin, b => aXb, c => sum);
   c3: AND PORT MAP (a => cin, b => aXb, c => abc);
   c4: AND PORT MAP (a => a ,  b => b,   c => ab);
   c5: OR  PORT MAP (a => ab,  b => abc, c => cout);
END fullAdder_structure;
```

(b) VHDL description

Figure 4.2: Translated VHDL description for the fullAdder example – structure

```
hw  fullAdder (cin, a, b, sum, cout) =
    sum := a xor b xor cin
 || cout := (a and b) or (cin and (a xor b))
```

(a) HML behavioral description of a 1-bit full adder

```
ENTITY fullAdder IS
   PORT( cin  : IN   bit;
         a    :  IN   bit;
         b    : IN   bit;
         sum  : OUT  bit;
         cout : OUT  bit);
END fullAdder;


ARCHITECTURE fullAdder_behavior OF fullAdder IS
   SIGNAL aXb, ab, abc;

BEGIN
   p1: PROCESS(a,b,cin)
   BEGIN
     sum <= a xor b xor cin;
   END PROCESS p1;

   p2: PROCESS(a,b,cin)
   BEGIN
     cout <= (a and b) or (cin and (a xor b));
   END PROCESS p2;
END fullAdder_behavior;
```

(b) VHDL description

Figure 4.3: Translated VHDL description for the fullAdder example – behavior

```
hw sum2 (a, b, s1, s2) =
       add (a, b, s1) (*structural format*)
    || s2 <- s1       (*behavioral format*)
```

(a) HML description

```
ENTITY sum2 IS
   PORT( clk: IN    bit;
 a:   IN    integer;
         b:   IN    integer;
         s1:  INOUT integer;
         s2:  OUT   integer);
END sum2;

ARCHITECTURE sum2_behavior OF sum2 IS
   COMPONENT add
      PORT(a: IN integer;
           b: IN integer;
           c: OUT integer);
   END COMPONENT;

BEGIN
   c1: add PORT MAP (a => a, b => b, c => s1);

   p2: PROCESS
   BEGIN
      WAIT ON clk;
      IF (clk='1' AND clk'LAST_VALUE='0' AND clk'EVENT) THEN
         s2 <= s1;
      END IF;
   END PROCESS p2;
END sum2_behavior;
```

(b) VHDL description

Figure 4.4: Translated VHDL description for an adder with output latch (mixture of structural and behavioral description)

4.2.2 Translation of Behaviors: Concurrent vs. Sequential Constructs

Inside an HML hardware function, the major part is the behavior expression, which consists of one or several behaviors. Translation of these behaviors involves the issue of distinguishing between concurrent and sequential constructs. As we have discussed in Section 2.3.3.3, concurrent and sequential operators "||" and ";" have the same hardware meaning but different writing styles.

The HML behaviors composed by operator "||" run concurrently; they form the top level behavior expressions. On the other hand, inside each HML behavior, all the hardware description expressions (composed by operator ";") are evaluated sequentially in zero time in each evaluation cycle.

These features of HML behaviors are like those of VHDL processes: different processes in an architecture execute concurrently, while inside a process, the statements are evaluated sequentially in zero time. Therefore, we translate an HML behavior into a VHDL process. The example of the simple adder with output latch in Figure 2.8 has two concurrent behaviors; it is translated automatically into the VHDL description shown in Figure 4.5.

Note that if "||" is used to compose submodules in structural HML descriptions, each submodule is translated into a VHDL component port map, not a process.

VHDL processes are always active if not suspended. A mechanism for conditionally activating a process is the use of a *sensitivity list*, a list of signals that can activate the process if an event occurs on any of these signals. HML behaviors do not need to specify a sensitivity list because all the information is included inside the behavior and can be inferred. For a process that is controlled by clock signals, the process is only active when there is a clock edge; for a process that is purely combinational, all the inputs of the process are included in its *sensitivity list*. In Figure 4.5, process `p1` has a sensitivity list of `a` and `b`; `p2`'s sensitivity list is the

```
hw sum (a, b, s1, s2) =
       s1 := a + b
    || s2 <- s1
```

(a) HML description of a simple adder

```
ENTITY sum IS
   PORT( clk: IN    bit;
 a:   IN    integer;
         b:   IN    integer;
         s1:  INOUT integer;
         s2:  OUT   integer);
END sum;

ARCHITECTURE sum_behavior OF sum IS
BEGIN
   p1: PROCESS(a,b)
   BEGIN
      s1 <= a + b;
   END PROCESS p1;

   p2: PROCESS
   BEGIN
      WAIT ON clk;
      IF (clk='1' AND clk'LAST_VALUE='0' AND clk'EVENT) THEN
         s2 <= s1;
      END IF;
   END PROCESS p2;
END sum_behavior;
```

(b) VHDL description

Figure 4.5: Translated VHDL description for the simple adder sum

`clk` signal and is specified with a `WAIT` statement.

4.2.3 Translation of Signal Assignments

HML has two different assignments for signals: combinational and register assignment. A combinational assignment updates the signal immediately; it is translated into a signal assignment in VHDL. A register assignment updates the signal at the next clock cycle; it is translated into a signal assignment controlled by a clock. Translation of signal assignments is shown in Figure 4.6.

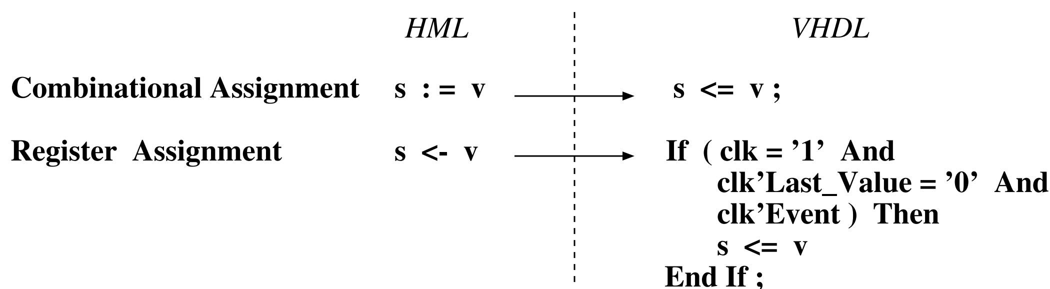

Figure 4.6: Translation of signal assignments

Because of this feature, an HML program does not necessarily include clocking information even for sequential circuits. As long as a register signal assignment appears in a behavior, it implies that the behavior is sequential. The translator will add clock signals to the process into which the behavior is translated, and prompt the user for the kind of clock they wish to use at compile time. The user can also choose to have a global reset signal. In the example of Figure 4.6, a rising edge clock is added.

4.2.4 Adding Declarations in Translation

As discussed, HML does not require users to specify types and interfaces. VHDL has a very verbose type system; all the objects have to have specified types, and for those used in entities as ports, interfaces must be specified. Therefore the type

and interface information obtained by the type-checker is added to the VHDL description.

In addition to adding type and interface specifications and directly translating the HML *val*, *type* and *intern* declarations, declarations have to be added in the following cases:

1. Entity declarations are added for each hardware function declaration, as discussed in Section 4.2.1.

2. Component declarations are added into the architecture body declarative part while translating a structural HML description. As in Figure 4.2, components `XOR`, `AND` and `OR` are declared before they are instantiated in the architecture body.

3. The declarations in *let-in-end* expressions should also be extracted and put in the appropriate location. HML's *let-in-end* expression provides a convenient way of declaring local objects. Since *let-in-end* is treated as an expression and not a declaration, it can appear anywhere in hardware expressions and can even be nested. VHDL does not have such a flexible construct, so the declarations that appear in *let-in-end* expressions must be popped to a higher level in VHDL. Usually these declarations are placed at the declaration part of the architecture body or that of the process.

4. HML has a shorthand for function declarations: anonymous functions defined by *fn* expressions. When translating to VHDL, this kind of function has to first be assigned a name, and then translated into a VHDL function declaration and added at an appropriate location. Figure 4.7 shows an example of a 4-bit counter, implemented with an increment function. Note that it can be written without the function; the function is added purely for the purpose of

illustrating function translation. Part (a) and (b) of Figure 4.7 are the HML program using *fn* notation and using a regular function declaration. Part (c) is the automatically translated VHDL program for both (a) and (b).

4.2.5 Translation for Simulation and Synthesis

The translator may generate different VHDL code from the same HML program. If VHDL is used only for functional simulation, the HML code is translated into VHDL without doing any optimization and the VHDL code is not necessarily synthesizable. On the other hand, if the VHDL code is used in synthesis, the translator does some simple optimizations (e.g. replace multiplication and division with shift if possible) and generates synthesizable VHDL.

Figure 4.8 part(a) gives a fragment of an HML description which contains a division in it. Included are also the translated VHDL descriptions for both the simulation (part(b)) and the synthesis version (part(c)).

The synthesis version of VHDL is the same as that of simulation except that:

1. If one operand is a constant of power of two, *Multiplications* and *divisions* are replaced by arithmetic *shifts* (division itself is not synthesizable by *AutoLogic*, the Mentor Graphics synthesis tool).

2. The *integer* type is transformed into *bit_vector* if the HML description has multiplications or divisions in it, because shift operations can only apply to bit_vectors in *AutoLogic* (this can also be done by the Mentor Graphics tool).

3. Some bidirectional ports are changed to single direction ports (`OUTPUT`); internal bidirectional signals are added as a replacement. At the end, the ports are assigned to the internal signals. For example, in Figure 4.8 in the synthesis version of VHDL, bidirectional port `a` is changed into an output; an internal signal `a_in` is added to replace `a`. The reason for this change is

```
hw counter (a) =
    ( a <- (fn x => if x = 15 then 0 else x  + 1) (a) )
```

(a). HML program using *fn* notation

```
hw counter (a) =
    let fun inc1 x = if x = 15 then 0 else x  + 1
    in ( a <- inc1 (a) )
    end
```

(b). HML program using regular function declaration

```
ENTITY counter IS
   PORT (clk : IN bit;
         a   : INOUT integer);
End counter;

ARCHITECTURE counter_behavior OF counter IS
   FUNCTION inc1 ( x : IN integer) RETURN integer IS
      variable result : integer;
   BEGIN
      IF result = 15 THEN
        result := 0;
      ELSE
        result := result + 1;
      END IF;
      RETURN result;
   END inc1;

BEGIN
   p1: PROCESS
   BEGIN
      WAIT ON clk;
      IF (clk='1' AND clk'LAST_VALUE='0' AND clk'EVENT) THEN
        a <= inc1 (a);
      END IF;
   END PROCESS p1;
END counter_behavior;
```

(c). Translated VHDL description for (a) and (b)

Figure 4.7: Translation of functions: Example of a 4-bit counter

```
hw example(a, ...) =
   ( ... ...
     a <- a / 4;
     ... ...)
```

(a) HML description

```
Entity example_sim IS
  PORT(clk: IN bit;
       a: INOUT integer;
       ... );
END example_sim;

ARCHITECTURE sim_behavior
       OF example_sim IS
    ... ...


BEGIN
  p1: PROCESS
  BEGIN
    WAIT ON clk;
    ... ...
    IF (clk='1' AND
        clk'LAST_VALUE='0' AND
        clk'EVENT) THEN
      a <= a /4;
    END IF;
    ... ...
  END PROCESS p1;



END sim_behavior;
```

(b) VHDL – simulation version

```
Entity example_syn IS
  PORT(clk: IN bit;
       a: OUT bit_vector(3 DOWNTO 0)
       ... );
END example_syn;

ARCHITECTURE syn_behavior
       OF example_syn IS
   ... ...
   SIGNAL a_in:
             bit_vector(3 DOWNTO 0);
BEGIN
  p1: PROCESS
  BEGIN
    WAIT ON clk;
    ... ...
    IF (clk='1' AND
        clk'LAST_VALUE='0' AND
        clk'EVENT) THEN
      a_in <= a_in sra 2;
    END IF;
    ... ...
  END PROCESS p1;

  a <= a_in;

END syn_behavior;
```

(c) VHDL – synthesis version

Figure 4.8: Different translation for simulation and synthesis

to eliminate bidirectional ports. *AutoLogic* requires that bidirectional ports be `BUS`es, which makes them become *guarded* signals. Since guarded signals have to be assigned in a *Block*, they would increase the complexity of the VHDL generated.

Users can choose between the two options by setting the synthesis flag in the translation command line; the default is to generate VHDL for simulation. Why don't we simply generate all synthesizable code? Because of the differences listed above,

1. synthesis version VHDL programs are usually slightly harder to read than the simulation programs,
2. synthesis version VHDL programs take slightly longer to generate with the HML-to-VHDL translator, and
3. synthesis version VHDL programs take slightly longer to simulate on VHDL simulation tools.

4.2.6 Other Issues in the Translation

Other than the issues that were addressed in previous sections, there are some relatively less prominent but still important issues to take into consideration.

- **HML expressions vs. VHDL statements:** In HML, expressions are used in hardware behaviors. Constructs such as signal assignments, *if-then-else*, *case* and *let-in-end* all are expressions. In VHDL, the architecture are formed by statements. HML expressions are much more flexible than VHDL statements, for example,

  `s := if condition then v1 else v2` and

  `if condition then s := v1 else s := v2`

are equivalent and valid HML expressions. But In VHDL, only the latter format is acceptable:

```
IF condition THEN s <= v1; ELSE s <= v2; END IF;
```

Therefore the translator can not just do a simple syntax mapping, it must identify different formats of HML expressions and translate them properly. Also, all VHDL statements are ended by semicolons; the translator adds semicolons when needed.

- **Object renaming:** Since all the basic types supported by HML are also supported by VHDL, the translation of objects is direct, except for names. HML allows name overloading of objects, and its flexible construct *let-in-end* makes it possible to define local objects almost anywhere. Therefore, it is possible for several objects with different scopes to share the same name. For example, in Figure 4.9 two `x`'s in the HML description have different scopes and values. But in VHDL, declarations can only appear at the declarative part of architecture, process, etc. The two declarations for `x` are both put in the declarative part of a process. In this case, renaming is needed. In fact, the translator renames all HML objects. The general methodology is to generate numbers by a sequence counter, and put the number at the end of each object name.

  In addition to renaming, there are some names that need to be created, including process names, component names and architecture names.

4.3 HML Features Not Implemented by HML-to-VHDL Translator

For most of the HML features that were addressed in Chapter 2, we have implemented their translation into VHDL in the HML-to-VHDL translator. However,

```
hw rename (a,b,c...) =
  ( ... ...
    let val x = 2
    in  b := a * x
    end;
    ... ...
    let val x = 4
    in  c := a * x
    end;
    ... ... )
```

(a) HML description

```
ENTITY rename IS ...
ARCHITECTURE ...
  ... ...
  p1 : PROCESS(a, ...)
    SIGNAL x1 : interger := 2;
    SIGNAL x2 : integer  := 4;
  BEGIN
    ... ...
    b <= a * x1;
    c <= a * x2;
    ... ...
```

(b) VHDL description

Figure 4.9: Object renaming in HML-to-VHDL translation

the following features are not implemented by the translator.

4.3.1 Recursive Functions

The feature of recursive functions is useful in describing regular structure generators. VHDL does not support recursive functions; recursions are simulated using *while* or *for* loops (although there are some kinds of recursions that can not be simulated by loops). We can restrict the recursions that could be used in HML and translation these recursions into VHDL loops. Note that *while-loops* are not supported by *AutoLogic* for synthesis.

4.3.2 High-order Functions

We have not implemented the translation of high-order functions. A possible way of translating high-order functions is to use structural VHDL to describe the high-order arguments as a component. This can be illustrated by the examples in Figure 4.10.

Figure 4.10 shows two structural compositions (sequential composition and parallel composition) of two cells *cell1* and *cell2*. Figure 4.11 gives the HML descriptions of the two compositions. In hardware functions `SeqComp` and `ParaComp`,

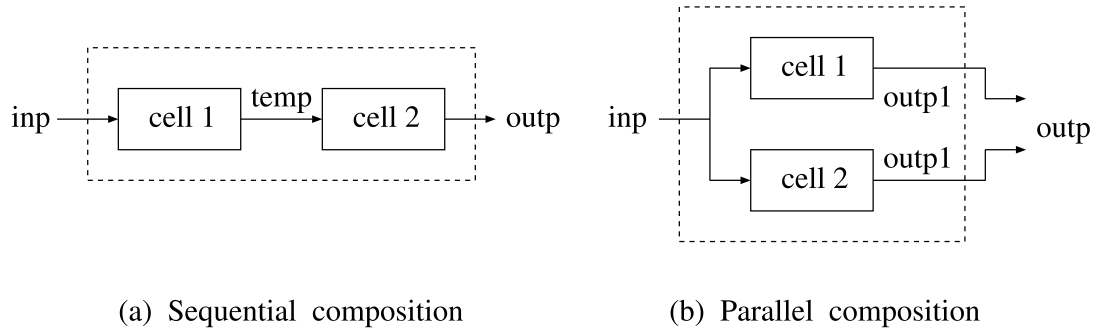

Figure 4.10: Two high-order structural compositions

arguments `cell1` and `cell2` are hardware functions themselves. They can have any internal behaviors with the interfaces of an input port and an output port.

Hardware functions `SeqComp` and `ParaComp` are general descriptions for composing two cells; they can not be translated into VHDL without specific information about the types of the two cells. Therefore the translation strategy is to generate VHDL description for each type of cell used. Assuming that `cell1` is of type

```
(in, bit-vector(3,0)) * (out, bit-vector(3,0) → behavior
```

and `cell2` is of type

```
(in, bit-vector(3,0)) * (out, bit-vector(7,0) → behavior
```

then `SeqComp` and `ParaComp` can be translated into VHDL, as shown in Figure 4.12 and Figure 4.13.

```
hw SeqComp (cell1, cell2, inp, outp) =
   let
      intern temp
   in
       cell1(inp, temp)
    || cell2(temp, outp)
   end
```

(a) HML description for sequential composition

```
hw ParaComp (cell1, cell2, inp, outp) =
    let
       intern  outp1, outp2
    in
          cell1(inp, outp1)
       || cell2(inp, outp2)
       || outp := outp1 @ outp2
end
```

(b) HML description for parallel composition

Figure 4.11: HML descriptions for two high-order structural compositions

```
ENTITY SeqComp IS
   Port( inp : IN  Bit_vector(3 downto 0);
         outp: OUT Bit_vector(7 downto 0));
END SeqCompl;

ARCHITECTURE SeqComp_arch OF SeqComp IS
   COMPONENT cell1
       PORT(a: IN bit_vector(3 downto 0);
            b: OUT bit_vector(3 downto 0));
   END COMPONENT;

   COMPONENT cell2
       PORT(a: IN bit_vector(3 downto 0);
            b: OUT bit_vector(7 downto 0));
   END COMPONENT;

   SIGNAL temp: bit_vector(3 downto 0);

BEGIN
   c1: cell1 PORT MAP (a => inp, b => temp);
   c2: cell2 PORT MAP (a => temp, b => outp);
END SeqComp_arch;
```

Figure 4.12: Translated VHDL descriptions for sequential compositions

```
ENTITY ParaComp IS
   Port( inp : IN  Bit_vector(3 downto 0);
         outp: OUT Bit_vector(11 downto 0));
END ParaComp;

ARCHITECTURE ParaComp_arch OF ConComp IS
   COMPONENT cell1
       PORT(a: IN bit_vector(3 downto 0);
            b: OUT bit_vector(3 downto 0));
   END COMPONENT;

   COMPONENT cell2
       PORT(a: IN bit_vector(3 downto 0);
            b: OUT bit_vector(7 downto 0));
   END COMPONENT;

   SIGNAL outp1: bit_vector(3 downto 0);
   SIGNAL outp2: bit_vector(7 downto 0);

BEGIN
   c1: cell1 PORT MAP (a => inp, b => outp1);
   c2: cell2 PORT MAP (a => inp, b => outp2);
   outp <= outp1 & outp2;
END ParaComp_arch;
```

Figure 4.13: Translated VHDL descriptions for parallel composition

4.4 Conclusion

This chapter discussed the translation from HML to VHDL. To summarize, the major considerations in the translation are:

1. To specify types and interfaces that are not specified in HML but are inferred by the type-checker.
2. To translate structural and behavioral descriptions appropriately.
3. To add/move HML declarations into appropriate places in VHDL.
4. To properly translate constructors and composers of HML into VHDL constructs.
5. To translate HML expressions into appropriate VHDL statements.
6. To rename the HML objects.
7. To incorporate clock and timing information if needed.
8. To translate differently for simulation and synthesis.

In Chapter 1 we have compared the two languages. Looking at the examples (both HML and VHDL descriptions) in this chapter, we can notice a significant difference between the programs of the two languages: the **length**. For the same example, the VHDL description is usually several times longer than the HML description. The comparison clearly shows HML's conciseness. **First**, this can definitely be attributed to HML's type system: there is no need to specify types or interfaces, while VHDL's type system is very verbose. **Second**, HML is concise due to its clean syntax for behaviors, functions and expressions. In VHDL, most constructs (e.g. *architecture, process, case expression, if expression*, etc.) have to be enclosed by `BEGIN` and `END`; HML's grammar is carefully designed so that

HML behaviors and expressions do not have to do so. Reviewing the example in Figure 4.7, a two-line HML program describes a behavioral hardware module even with a software function defined and used in it!

Chapter 5

Implementation in SML

The HML system that we have implemented includes a front-end HML parser, a type-checker which automatically infers types and interfaces and also checks the typing errors and some design rule errors, and an HML-to-VHDL translator which translates HML programs into synthesizable VHDL programs. The implementation is in SML of New Jersey (SML-NJ) [AT93]. This chapter explains the organization of the source programs, important data structures and functions.

5.1 Organization of HML Source Programs

The HML system has three major parts: parser, type-checker and the HML-to-VHDL translator. The HML source directory includes the following files:

```
base.sml,  ast.sml,  message.sml

hml.lex --> hml.lex.sml
hml.grm --> hml.grm.sig,  hml.grm.sml
join.sml

typedef.sml, typecheck.sml

vhdl.sml,  hml2vhdl.sml
```

`export.sml, all.sml`

Figure 5.1 illustrates the organization of these programs.

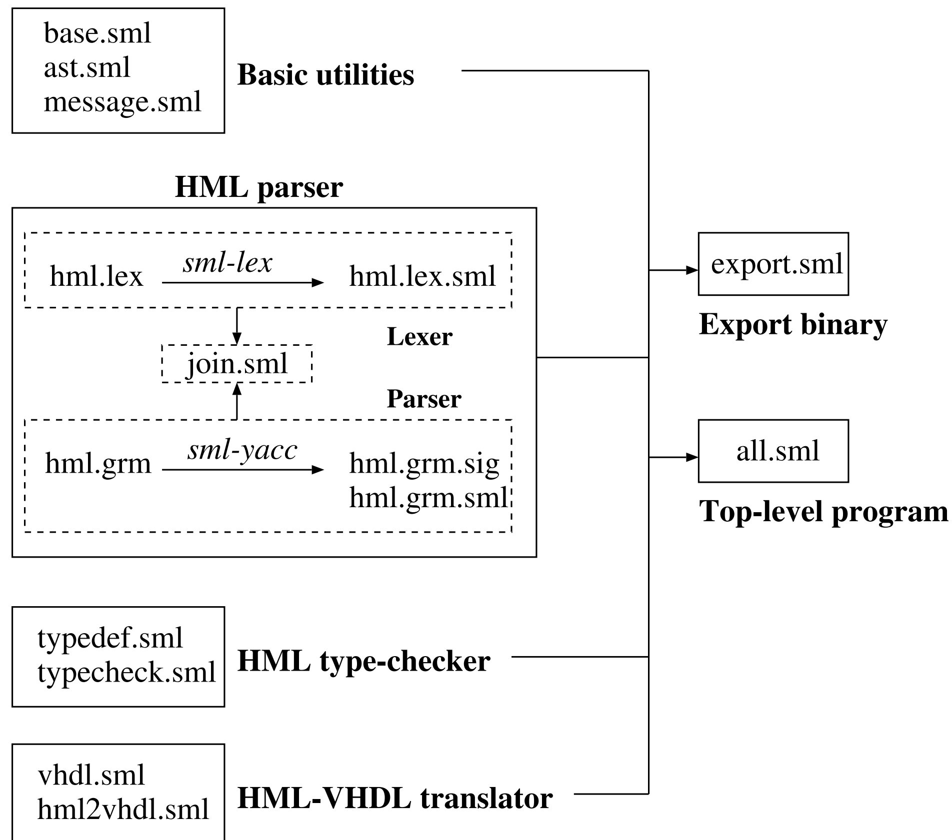

Figure 5.1: Organization of HML source programs

`Base.sml`, `ast.sml` and `message.sml` are the **basic utilities** that are used throughout the system. `Base.sml` is the base environment of SML-NJ and contains the common modules for the lexer and the parser; it also provides some useful functions. `Ast.sml` is the structure of the abstract syntax tree of HML, which is the basis of the whole system. The abstract syntax tree is defined by defining the nonterminals as datatypes. `Message.sml` defines functions for printing messages such as error messages.

The **HML parser** is generated by applying the SML lexer generating tool SML-lex [AT93] and the SML parser generating tool SML-Yacc [AT93]. `Hml.lex`

is the lexer program written in SML-Lex. SML-Lex produces an SML program `hml.lex.sml` from `hml.lex` , which is the HML lexical analyzer. `Hml.grm` defines HML LALR1 grammar rules in SML-Yacc. SML-Yacc creates two SML programs `hml.grm.sig` and `hml.grm.sml` as programs for the primitive parser.

To create the final parser, some functors must be applied. This is done by the program `join.sml` which joins the lexer and the primitive parser to create a parsing function called `parse`.

The **HML type-checker** mainly consists of two programs: `typedef.sml` and `typecheck.sml`. `Typedef.sml` is a structure that defines the data structures used in type-checking and some common functions that are used in both the type-checker and the HML-to-VHDL translator. `Typecheck.sml` is the major part of the type-checker; it defines the type-checking and type inference functions of HML based on the HML abstract syntax tree defined in `ast.sml`.

Programs `vhdl.sml` and `hml2vhdl.sml` form the HML-to-VHDL translator. `Vhdl.sml` defines the translation functions for all HML constructs; `hml2vhdl.sml` is the top-level code for both the type-checker and the HML-to-VHDL translator that defines the top-level type-checking and translation functions visible to users.

All the source programs can be loaded in an SML environment by simply loading program `all.sml` which includes all the source programs in the correct order. Program `export.sml` includes functions to export executable binary in *SML-export* [AT93]. If loaded into *SML-export*, it exports programs called `parse`, `typecheck`, and `hml2vhdl` which are the executable parser, type-checker and HML-to-VHDL translator respectively.

5.2 Data Structures and Functions

The HML system is built by functions grouped in different SML structures. There are some important data structures that are used across several structures and

in many functions. These data structures and the top-level functions of each structures are explained in this section. Refer to Appendix C for a list of the signatures for these structure.

Data Structures

- **Abstract syntax tree :** The HML *abstract syntax tree* is defined in file `ast.sml` as a structure consisting of datatypes. The datatypes are the defined-types for HML nonterminals, such as *dec* (declaration) and *exp* (expression), etc.

- **Data structures used in the parser :** SML-Yacc takes the HML grammar as input and produces structures and functors used in the parser. The functions and usage of these structures and functors are discussed in the SML-Yacc user's manual [AT93]. The parsing result uses the datatypes defined in the HML *abstract syntax tree* and the output is used as the interface to the type-checker and the HML-to-VHDL translator.

- **Environments :** In `typedef.sml`, a datatype called *env*(environment) is defined. It is a very important data structure used in the type-checker and as the interface of the type-checker and the HML-to-VHDL translator. It is defined as a dynamic list of objects used in the program. There are two kinds of environments: *global environment* and *local environment*. The *global environment* is used mainly for interfacing the type-checker to the translator; it keeps track of all the objects ever used in the program. While translating HML to VHDL, the translator simply looks up the global environment and gets the typing information needed in VHDL. The *local environment* changes dynamically; it only keeps the objects that are still needed by the programs and throws out the obsolete ones. It is mainly used in the type-checker. The global environment can be seen as the main output of the type-checker; with

the information in the global environment, the type-checker can be interfaced to various back-ends.

- **Object information :** Environments are stored as lists of objects. Each element of those lists records all the key information associated with an object. This includes the following features of an object: name, type, input/output, type decidability and object scope. These features are put into an SML record. Type decidability is a boolean representing whether the type of an object has been decided. Object scope is important because HML allows overloading of object names and the global environment has to keep all the objects used in the program so that it is possible to have two objects with the same name but different scopes.

- **Type information :** Type information is a tuple of *mode* (which is either constant or variable) and pure type.

Functions

- **HMLBatchParser** *file*, **parse** *file* **:** *HMLBatchparser* is the internal parsing function that takes a file name as input and returns the result in data structure *parse-result* (defined by SML-Yacc [AT93]). *Parse* is a user visible image of *HMLBatchparser*, it is exported to an executable file.

- **EnvOfDec** *dec*, **TopTypeOfDec** *dec* **:** These two functions are internal type-checking functions: *EnvOfDec* can do type checking on any declarations and updates the environments accordingly; *TopTypeOfDec* is based on *EnvOfDec* and does the type-checking of the top-level program which is a sequential declaration; it returns the *global environment* of the program.

- **typecheck** *file_name* **:** This is the user visible version of combined *parse* and *TopTypeOfDec*. It first parses the program *file_name*. Besides type-checking

and type inference, it prints all the error messages to standard output and puts the type-checking result in file *file_name.log*.

- **TopCodeOfDec, MainCodeOfDec, PackageCodeOfDec, PackageBodyCodeOfDec, CodeOfDec :** *MainCodeOfDec, PackageCodeOfDec* and *PackageBodyCodeOfDec* are functions that generate VHDL code from an HML declaration. The three functions generate three pieces of VHDL code that are the main code with VHDL entity and architecture, the VHDL package code, and the code for the package body. *TopCodeOfDec* puts the three functions together and returns a tuple of the three pieces of the VHDL code.

- **hml2vhdl** *file syn_flag clk_flag* : The is the user visible HML-to-VHDL translation function. The *syn_flag* indicates whether to generate VHDL code for synthesis. The *clk_flag* indicates whether to add clocks automatically. It takes a file name as input, first does the parsing and type-checking and then HML-to-VHDL translation. The VHDL code is save in files *file.vhd* (the main VHDL program with entity and architecture definitions) and *file.lib.vhd* (the VHDL package program).

Chapter 6

Examples

This chapter contains two examples: a description of an integer square root and an adder/subtracter ALU. Both the HML descriptions and the VHDL descriptions generated by the HML-to-VHDL translator are given, as well as the results of simulation/synthesis of VHDL descriptions on Mentor-Graphics tools.

6.1 Non-restoring Integer Square-root

This section describes how to implement a subtractive, non-restoring integer square root algorithm in HML and how it is translated to VHDL.

6.1.1 The Non-restoring Integer Square Root Algorithm

Definition: Correct integer square root

y is the correct *integer square root of x if* $y^2 \leq x < (y+1)^2$

An integer square root calculates $y = \sqrt{x}$ where x is the radicand, y is the root, and both x and y are integers. We define the *precise* square root (p) to be the real valued square root and the *correct* integer square root to be the floor of the precise

root.

We have implemented a subtractive, non-restoring square root algorithm [OLHA95]. Subtractive methods begin with an initial guess of $y = 2^{(n-1)}$ (assume that the input is a $2n$ bit integer) and then iterate from $i = (n - 1)$ *downto* 0. In each iteration we square the partial root (y), subtract the squared partial root from the radicand and revise the partial root based on the sign of the result. In binary arithmetic, each bit in the partial result is effectively modified only once.

The resulting value of y in the non-restoring algorithms is not correct because there may be an error in the last bit position. For the algorithm used here, we can show that the final value of y will always be either the precise root (for radicands which are perfect squares) or will be odd and be within one of the correct root.

We have shown how a non-restoring integer square root algorithm can be transformed to a very efficient hardware implementation [OLHA95]. The top level algorithm is an SML function that operates on unbounded integers. Figure 6.1 shows the level 2 algorithm written in SML, evolved from the top level SML algorithm.

6.1.2 Describing the Integer Square Root in HML

The SML algorithm described in Section 6.1.1 can be easily transformed into a behavioral hardware description in HML, shown in Figure 6.2.

This example assumes an 8-bit input. This is specified by declaration `width 7,0`, meaning the bit-vectors synthesized are labeled from 7 downto 0. `initb` is the initial guess of the result.

The major part of the program is the *hw* specification. Thanks to HML's concise syntax and that fact that types do not need to be specified, the program emphasizes the abstract behavior of the hardware, and therefore is almost a straight-forward translation of the level 2 SML algorithm shown in Section 6.1.1. Clock information

```
fun init2 (n,radicand) =
  State{diffx  = radicand,
        yshift = 0,
        b      = 2 ** (2*(n-1))}

fun update1 (State{diffx, yshift, b}) =
  let
    val (diffx',yshift')  =
       if      diffx > 0 then (diffx - yshift - b, yshift + 2*b)
       else if diffx = 0 then (diffx             , yshift      )
       else (* diffx < 0 *)   (diffx + yshift - b, yshift - 2*b)
  in
   State{diffx  = diffx',
         yshift = yshift' div 2,
         b      = b  div 4
        }
  end
```

Figure 6.1: Non-restoring integer square root Level 2 algorithm in SML

is omitted in the program. An major difference between Figure 6.1 and Figure 6.2 is the use of `init` and `Done` signals. In the SML description of the algorithm, the initiation and the termination of the algorithm are handled by the SML interpreter. Because hardware is "free running" there is no built in notion of initiation or termination of the algorithm. Therefore *init* and `Done` are added to explicitly initialize the algorithm and to detect when the algorithm has terminated.

In computing a four-bit result, the level 2 algorithm terminates after four iterations of its loop. An efficient way to detect termination of the hardware algorithm makes use of some knowledge of the high level algorithm. An informal analysis of the level 2 algorithm reveals that `B` contains a single bit, shifted right two places in each cycle, and that the '1' bit of `B` is shifted to the least significant bit before the execution of the last iteration and `B` becomes 1. Consequently, the `Done` signal is generated by testing whether the `B` is 1. `Done` is therefore set during the clock cycle following the final iteration. Because hardware is free running, the program

```
width 7,0                  (*8-bit input*)

val initb = 64            (*0x40*)

hw sqrt (Init, XIn, YShift, Done) =
  let
      val DiffX = 0
      val B = 0
  in
     if Init='1' then
         (DiffX <- XIn;
          YShift <- 0;
          B <- initb;
          Done <- '0')
      else
         if  Done='0' then
         ( if DiffX = 0  then
              (DiffX <- DiffX;
               YShift <- YShift div 2;
               B <- B div 4)
           else if DiffX > 0 then
              (DiffX <- DiffX - YShift - B;
               YShift <- (YShift +  B * 2) div 2;
               B <- B div 4)
           else (* DiffX < 0 *)
              (DiffX <- DiffX + YShift - B;
               YShift <- (YShift - B * 2) div 2;
               B <- B div 4) ;
           Done <- B[0] )
  end
```

Figure 6.2: HML description of integer square root Level 2 algorithm

will run through all the iterations until the terminating condition is satisfied and therefore `Done` is set.

6.1.3 VHDL Code and Simulation/Synthesis on Mentor-Graphics Tools

To simulate and synthesize the HML description in Figure 6.2, the HML-to-VHDL translator is called to translate the HML code into VHDL. Figure 6.3 shows the generated VHDL code for simulation.

The global `val` declaration in the HML code is put into the package of VHDL. An integer type declaration is added to limit the bus width. An VHDL entity for `sqrt` is added based on the type checking information.

A `clk` signal is added because there are register assignments in the original HML code. Note that this `clk` is not necessarily the clock that is used in the actual circuit, since the multiplications and divisions inside the process might take one or more cycles(synthesized as shift operations). However, this simplified clock signal is enough for a functional simulation. Other translation is direct mapping according to the rules described in Table 4.2. Refer to the comments in the VHDL code. The comments were added by hand.

Mentor Graphics V8.4_1 software *QuickVHDL* [Men94b] is used to simulate the VHDL program. The simulation stimulus file is shown in Figure 6.4. Two input values **49** and **28** are simulated. Figure 6.5 shows the simulation waveforms of all the signals. It is clear that after 4 iterations, we get the correct square roots of `Xin`, i.e. **7** and **5**.

To synthesize the HML description, the translator is called with the synthesis flag set. The generated VHDL specification (Figure 6.6) has the following difference from the simulation version:

1. *Multiplications* and *divisions* are replaced by arithmetic *shifts*.

```
ENTITY sqrt IS     -- sqrt.vhd
    PORT( clk     : IN bit;
          Init    : IN bit;
          Xin     : IN integer;
          YShift  : INOUT integer;
          Done    : INOUT bit);
END;

ARCHITECTURE  sqrt_behavior OF sqrt IS
    SIGNAL DiffX: integer :=0;
    SIGNAL B : integer := 0;
BEGIN
  p1: PROCESS(clk)
  BEGIN
    IF (clk='1' AND clk'EVENT AND clk'LAST_VALUE='0')THEN  -- clk added
      IF (Init = '1') THEN
        DiffX <= Xin;
        Yshift <= 0;
        B <= initb;
        Done <= '0';
      ELSE IF (Done = '0')THEN
        IF (DiffX = 0) THEN
           DiffX <= DiffX;
           Yshift <= Yshift / 2;
           B <= B / 4 ;
        ELSE IF (DiffX > 0) THEN
           DiffX <= DiffX - Yshift - B;
           Yshift <= (Yshift + 2 * B) / 2;
           B <= B /4;
        ELSE
           DiffX <= DiffX + Yshift - B;
           Yshift <= (Yshift - 2 * B) / 2;
           B <= B / 4 ;
        END IF;
        END IF;

        Done <= B(0);

        END IF;
      END IF;
    END IF;
  END PROCESS p1;
END sqrt_behavior;   -- end of sqrt.vhd
```

Figure 6.3: VHDL description of integer square root produced by VHDL-HML translator – Simulation Version

```
force clk 1 50, 0 100 -repeat 100
force init 1 50, 0 100, 1 600, 0 700
force Xin 0 0, 49 50, 28 550
run 1200
```

Figure 6.4: QuickVHDL simulation stimulus file of square root example

Figure 6.5: Simulation wave form of square root example

```
ENTITY sqrt IS
    PORT(   clk     : IN bit;
            Init    : IN bit;
            Xin     : IN  bit_vector(7 DOWNTO 0);
            YShift  : OUT bit_vector(7 DOWNTO 0);
            Done    : OUT bit);
END;

ARCHITECTURE  sqrt_behavior OF sqrt IS
        signal DiffX:    bit_vector(7 DOWNTO 0) := "00000000";
        signal B :       bit_vector(7 DOWNTO 0) := "00000000";
        signal Yshift_in:bit_vector(7 DOWNTO 0);
        signal Done_in:  bit;
BEGIN
    p1: process(clk)
    BEGIN
      IF (clk='1' AND clk'EVENT AND clk'LAST_VALUE='0') THEN
        IF (Init = '1') THEN
            DiffX <= Xin;
            Yshift_in <= "00000000";
            B <= initb;
            Done_in <= '0';
        ELSE IF (Done_in = '0') THEN
            IF ( DiffX = "00000000") THEN
                  DiffX <= DiffX;
                  Yshift_in <= Yshift_in sra 1;
                  B <= B sra 2 ;
            ELSE IF (DiffX(7) > "00000000") THEN
                  DiffX <= DiffX - Yshift_in - B;
                  Yshift_in <=  (Yshift_in sra 1) + B;
                  B <= B sra 2;
            ELSE  DiffX <= DiffX + Yshift_in - B;
                  Yshift_in <= (Yshift_in sra 1) - B ;
                  B <= B sra 2;
            END IF;  END IF;
            Done_in <= B(0);
        END IF;  END IF;
      END IF;
    END PROCESS p1;
        Yshift <= Yshift_in;
        Done <= Done_in;
END sqrt_behavior;
```

Figure 6.6: VHDL description of integer square root produced by VHDL-HML translator – Synthesis Version

2. The *integer* type is transformed into *bit_vector*.

3. `Yshift` and `Done` are changed to single direction signals. To keep the functionality two internal bidirectional signals, `Yshift_in` and `Done_in`, are added. In the end, `Yshift` and `Done` are assigned from `Yshift_in` and `Done_in`.

6.2 Adder/Subtracter ALU

The design in this section is an adder/subtracter ALU with input/output latches and result registers. It is controlled by signals `start, do_add, do_subtract,` and `do_hold` to decide whether to reset, to do addition/subtraction, or to hold the result. Section 6.2.1 presents the HML description for the design. Section 6.2.2 discusses the generated VHDL code for the synthesis version.

6.2.1 Describing Adder/Subtracter ALU in HML

To write well-structured HML programs, a complicated specification of a design can be partitioned into several behaviors. Each behavior represent a submodule in a design. According to the specification of the adder/subtracter ALU, we partition it into five parts: input latch, output latch, a state machine, result registers and the core ALU (combinational part). Each part of the ALU functionality is represented by a behavior expression and all the behaviors are grouped together using the concurrent operator "||". The HML description is shown in Figure 6.7.

6.2.2 Generated VHDL Description of Adder/Subtracter ALU

Using the HML-to-VHDL translator, the VHDL description is generated based on the HML description in Section 6.2.1 (see Figure 6.8, Figure 6.9 and Figure 6.10).

This design was used in the *Mentor-Graphics AutoLogic Synthesis Guide* [Men94a] as an example of writing VHDL programs for synthesis. The VHDL description

```
width 3,0       (* set the bit-vector width to be 4 *)
hw  add_sub_alu(
    rst,                                (*reset*)
    enable_in, enable_out,              (*control of data input/output*)
    start, do_add, do_subtract, do_hold,(*control of state*)
    data_in, data_out  ) =              (*data input and output*)
let
   type  states = hold | reset |  add | subtract  (*define states type*)
   intern state_var, reg,int_reg, latched_data_in (*internal signals*)
in
    (*1. data input*)  if enable_in = '1' then
                          latched_data_in <- data_in

|| ((*2.fsm: set state_var according to control inputs & current state*)
    if (rst = '1') then
        state_var <- reset
    else
      case state_var of
        hold =>     if (start = '1') then
                        state_var <- reset
      | reset =>    if (do_add = '1') then
                        state_var <- add
                     else if  (do_subtract = '1') then
                        state_var <- subtract
      |  add =>     if  (do_hold = '1') then
                       state_var <- hold
                    else if  (do_subtract = '1') then
                            state_var <- subtract
      | subtract =>if (do_hold = '1') then
                            state_var <- hold
                    else if  (do_add = '1') then
                            state_var <- add
        )
|| ((*3. alu *)  case  state_var of
                    add =>      int_reg := reg + latched_data_in
                 |  subtract => int_reg := reg - latched_data_in
                 |  reset =>    int_reg := "0000"
                 |  hold =>     int_reg := reg
    )
|| (*4. mem: store result of alu *)  reg <- int_reg

|| (*5. output data*)  if (enable_out = '1') then
                           data_out <- reg
end
```

Figure 6.7: HML description of an adder/subtracter ALU

```
-- Start of file "alu.lib.vhd"
-- Package
PACKAGE t_alu_lib  IS
   TYPE subinteger IS RANGE -128 TO 127 ;
END t_alu_lib;

-- Empty Package body
PACKAGE BODY t_alu_lib IS
END t_alu_lib;
-- End of file "alu.lib.vhd"

-- Start of file "alu.vhd"
LIBRARY WORK;
USE WORK.t_alu_lib.ALL;
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.ALL;

-- Entity : the interface
ENTITY add_sub_alu IS
    PORT(
       clk :          IN  BIT ;
       rst :          IN  BIT ;
       enable_in :    IN  BIT ;
       enable_out :   IN  BIT ;
       start :        IN  BIT ;
       do_add :       IN  BIT ;
       do_substract : IN  BIT ;
       do_hold :      IN  BIT ;
       data_in :      IN  BIT_VECTOR ( 3 DOWNTO 0 );
       data_out :     OUT BIT_VECTOR ( 3 DOWNTO 0 )
        );
END add_sub_alu;

-- Architecture
ARCHITECTURE add_sub_alu_behavior OF add_sub_alu IS
      TYPE states IS ( hold,reset,add,substract );

      SIGNAL state_var :       states ;
      SIGNAL reg :             BIT_VECTOR ( 3 DOWNTO 0 );
      SIGNAL int_reg :         BIT_VECTOR ( 3 DOWNTO 0 );
      SIGNAL latched_data_in : BIT_VECTOR ( 3 DOWNTO 0 );
BEGIN
```

Figure 6.8: VHDL description of the adder/subtracter ALU produced by VHDL-HML translator (Part 1, to be continued)

```
p1 : PROCESS     -- (*1. data input*)
BEGIN  WAIT ON clk  ;
   IF (clk='1' AND clk'LAST_VALUE='0' and clk'EVENT) THEN
      IF enable_in =  '1' THEN
         latched_data_in <= data_in ;
      END IF  ;
   END IF ;
END PROCESS p1 ;

p2 : PROCESS    -- (*2. fsm*)
BEGIN   WAIT ON clk  ;
   IF (clk='1' AND clk'LAST_VALUE='0' and clk'EVENT) THEN
      IF ( rst =  '1' )  THEN  state_var <= reset ;
      ELSE
         CASE state_var IS
           WHEN hold =>  IF ( start =  '1' ) THEN
                            state_var <= reset ;
                         END IF ;
           WHEN reset => IF ( do_add =  '1') THEN
                            state_var <= add ;
                         ELSE
                            IF ( do_substract =  '1') THEN
                               state_var <= substract ;
                            END IF ;
                         END IF ;
           WHEN add =>   IF ( do_hold =  '1')THEN
                            state_var <= hold ;
                         ELSE
                            IF ( do_substract =  '1') THEN
                              state_var <= substract ;
                            END IF  ;
                         END IF ;
           WHEN substract => IF ( do_hold = '1') THEN
                                state_var <= hold ;
                             ELSE
                                IF ( do_add =  '1') THEN
                                   state_var <= add ;
                                END IF  ;
                             END IF ;
         END CASE  ;
      END IF  ;
   END IF ;
END PROCESS p2 ;
```

Figure 6.9: VHDL description of the adder/subtracter ALU produced by VHDL-HML translator (Part 2, continued)

```
p3 : PROCESS (state_var, latched_data_in, reg)  --(*3. alu *)
BEGIN
   CASE state_var IS
      WHEN add       => int_reg <= reg + latched_data_in;
      WHEN substract => int_reg <= reg - latched_data_in;
      WHEN reset     => int_reg <= "0000";
      WHEN hold      => int_reg <= reg;
   END CASE  ;
END PROCESS p3 ;

p4 : PROCESS     -- (*4. mem: store result of alu *)
BEGIN
   WAIT ON clk  ;
   IF (clk = '1' AND clk'LAST_VALUE = '0' and clk'EVENT) THEN
      reg <= int_reg ;
   END IF ;
END PROCESS p4 ;

p5 : PROCESS    -- (*5. output data*)
BEGIN
   WAIT ON clk  ;
   IF (clk = '1' AND clk'LAST_VALUE = '0' and clk'EVENT) THEN
      IF  ( enable =  '1' )  THEN
         data_out <= reg ;
      END IF  ;
   END IF ;
END PROCESS p5 ;

END add_sub_alu_behavior ;
```

Figure 6.10: VHDL description of the adder/subtracter ALU produced by VHDL-HML translator (Part 3, continued)

generated is about the same as that give by the *AutoLogic Synthesis Guide*. Comparing the VHDL description and the HML description, it is quite obvious that the VHDL description is significantly longer that the HML description.

The length difference is caused by several factors: VHDL has an entity declaration which declares the interface while HML does not need to do so; in VHDL all the signals have to be declared before use; HML's clock information is implied by register assignment "$<-$" while in a VHDL process, clock information has to be described by an *IF* statement; and VHDL has very verbose syntax for its programming constructs: most constructs are enclose by `BEGIN` and `END`. This example again shows HML's conciseness.

Chapter 7

Conclusions and Future Plans

7.1 Conclusions

This thesis presents the HML language and describes how it can be translated to VHDL and used with tools for the latter language. We believe HML has many advantages over existing HDLs. These includes:

1. **HML's advanced type system** makes HML unique compared with other hardware description languages. Firstly, HML's polymorphic functions encourage code reuse since structure generators and commonly used modules can be shared among multiple types and multiple designs. Secondly, the automatic type inference allows users to write descriptions without specifying types and interfaces, therefore users can focus on functionality. Thirdly, strong type checking is able to flag many design rule violations early in the design process. Finally, HML as a high-order language provides an abstract and high-level notation for hardware structures.

2. **HML has extremely concise syntax** and is very easy to read and write. For the same design, most of the other HDLs that have been compared with HML can not gives a shorter or cleaner description than HML. This is due

to the type system that does not require type and interface specification. More importantly, it is because of the carefully designed grammar that HML descriptions are very succinct.

3. **HML has an open back-end:** Our methodology provides an HML front end (including that parser and the type checker), which can be linked to different back ends for use with other tools. The implementation of **the interface to VHDL** make it possible to take advantage of the wide variety of available VHDL tools. Generating **synthesizable VHDL** is also attractive: users can write HML hardware descriptions which can be synthesized and can also be easily **integrated with other VHDL designs**. Other back-ends might be an HML behavior simulator, an HML behavior synthesizer or a verifier.

7.2 Future Work

Future work includes continuing to test the current system and improving the language and the system. HML should be applied to more examples in order to debug the current system. Future improvements could include:

- Specifying a formal semantics and improving the formal definition of the language.
- Adding more features such as multiple clocks, hardware combinators, and implementing dependent types.
- Building HML's own behavior simulator: Since HML only uses a subset of VHDL and there are some unique features of HML that are not supported by VHDL, we believe an HML simulator could be much smaller and faster that the current *Mentor-Graphics* VHDL simulator that we are using. In

addition, not needing to do the HML-to-VHDL translation can shorten the simulation cycle.

- Building library support: HML has a small predefined library of basic gates, this ought to be expanded to include more useful functions and hardware modules.

- Embedding HML within theorem provers. HML's description is similar to structural descriptions in higher order logic theorem provers such as HOL [BGG+92] and Nuprl [Lee92]. The Computer Science Department of Brigham Young University is already doing some work on embedding HML in the HOL system.

- Improving the pretty printing functions in the HML-to-VHDL translator: The current translator performs some pretty printing while doing the translation, but it is very limited and should be improved.

Appendix A

HML Grammar

This appendix contains the grammar of HML. The grammar is LALR1 and can be used with SML-Yacc. Names in lower-case are nonterminals, names in upper-case are HML tokens.

```
(*------------- program --------------*)
program : sdecs

(*--------------- decs ---------------*)
(* global decs *)
sdecs   : sdec sdecs
        | sdec

sdec    : VAL vb
        | FUN fb
        | TYPE tb
        | HW hb
        | RANGE INTEGER0 COMMA INTEGER0
        | WIDTH INTEGER0 COMMA INTEGER0

(* local decs, including INTERN *)
ldec    : VAL vb
        | FUN fb
        | TYPE tb
        | INTERN pat_1c
```

```
ldecs :
        | ldec ldecs


(*--------------- vb -----------------*)
vb   :  pat EQUAL exp

constraint :
           | COLON ty

(*--------------- fb -----------------*)
fb      : clauses

clauses : clause
     | clause BAR clauses

clause  : ID pats constraint EQUAL exp

(*-------------- hb ------------------*)
hb      : hclauses

hclauses: hclause
     | hclause BAR hclauses

hclause : ID pats constraint EQUAL hwexp

(*---------------- tb ----------------*)
tb      : ID EQUAL ty_dec

ty_dec  : ty
        | id_2list

ty      : INT
        | BOOL
        | BITVECTOR LBRACKET INTEGER0 COMMA INTEGER0 RBRACKET
        | BIT
        | UNIT
        | LPAREN ty RPAREN
        | ID

(* used in EnuTy *)
id_2list: ID BAR ID
        | ID BAR id_2list
```

```
(*---------- match, rule, pat --------*)
match   : rule
        | rule BAR match

rule    : pat DARROW exp

constraint_pat : ID COLON ty

pat     : WILD
        | INTEGER
        | INTEGER0
        | HIGH
        | LOW
        | TRUE
        | FALSE
        | ID
        | LPAREN RPAREN
        | LPAREN pat RPAREN
        | LPAREN constraint_pat RPAREN

pat_1c  : pat
        | pat COMMA pat_1c

(* use in fun-decl as parameters *)
pats    : pats'
        | LPAREN pat_1c RPAREN

pats'   : pat
        | pat pats


(*------------- exp ----------------*)
hwexp   : beha_exp
       | stru_exp
        | LET ldecs IN beha_exps END
      | LET ldecs IN aexp END
        | LET ldecs IN stru_exp END

beha_exp : aexp
         | beha_exps

beha_exps : exp DOUBLEBAR exp
          | exp DOUBLEBAR beha_exps
```

```
stru_exp : STRUCTURE LPAREN app_exps RPAREN

app_exps : app_exp
         | app_exp SEMICOLON app_exps

option_exp :
           | COMMA exp

else_exp:
        | ELSE exp

exp     : aexp
        | let_exp

aexp    : LPAREN exp_2ps  RPAREN
        | LPAREN exp RPAREN
        | INTEGER0
        | INTEGER
        | HIGH
        | LOW
        | TRUE
        | FALSE
        | VECTOR
        | id_exp LBRACKET exp option_exp RBRACKET
        | LPAREN RPAREN   (* unit exp *)
        | pat_exp COLONEQUAL exp
        | pat_exp LARROW exp

        | exp ORELSE exp
        | exp ANDALSO exp
        | NOT exp
        | exp ADD exp
        | exp SUB exp
        | exp MUL exp
        | exp DIV exp
        | exp GT exp
        | exp GE exp
        | exp LT exp
        | exp LE exp
        | exp NE exp
        | exp EQUAL exp
        | exp AND exp
        | exp OR exp
```

```
        | exp NAND exp
        | exp NOR exp
        | exp XOR exp
        | exp XNOR exp
        | INV exp
        | IF exp THEN exp else_exp
        | CASE exp OF match
        | fn_exp
        | pat_exp
        | app_exp

let_exp : LET ldecs IN exp_1ps END

pat_exp : id_exp

fn_exp  : FN match

id_exp  : ID

app_exp : id_exp  exps
        | LPAREN fn_exp RPAREN  exps

exp_2ps : exp SEMICOLON exp
        | exp SEMICOLON exp_2ps

exp_1ps : exp
        | exp SEMICOLON exp_1ps

exps    :  LPAREN exp_1c RPAREN

exp_1c  : exp COMMA exp_1c
        | exp
```

Appendix B

HML2VHDL User's Manual

This appendix explains how to use the HML system. For details about how to write programs in HML, refer to Chapter 2 and Chapter 6.

The HML package includes an HML parser, a type checker and an HML-to-VHDL translator. It provides three executable files:

1. parse : This is the parser only. If invoked, it will check the syntax of the HML program. The format for the parse command is:

`parse` *filename*

It prints out syntax errors and does not have an output.

2. type check : This is the combined parser and type checker. It invokes the parser first and if there are no syntax errors it does the type checking and type inference, prints out the type checking errors on standard output if there are any, and exports the type checking and type information to file "`hml_file_name.log`". The format for the type check command is:

`typecheck` *filename*

3. hml2vhdl : This includes all three stages in our HML system: parsing, type checking and then translation to VHDL. It doesn't proceed to the next stage if there are errors at the current stage. The format for this command is:

`hml2vhdl [-syn] [-clk]` *filename*

There are two optional flags: if the `-syn` flag is set, the translator generates VHDL code synthesizable by Mentor tools, otherwise the default is to generate VHDL for simulation. The `-clk` flag will add clock information automatically while translating the HML program into VHDL. The `-clk` flag should be set only if the HML program doesn't specify clocking.

`hml2vhdl` produces three output files: `filename.log` is the type checking log file, it lists the type information of all the objects used in the HML program `filename`. `filename.vhd` is the main VHDL file that includes the entity declaration and architecture body; `filename_lib.vhd` is the package file for `filename.vhd`. When doing the simulation or synthesis, `filename_lib.vhd` should be compiled before `filename.vhd` by a VHDL compiler.

Bibliography

[Ash90] Peter J. Ashenden. *The VHDL Cookbook*. 1990.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AT93] AT&T Bell Laboratories. *Standard ML of New Jersey - User's Guide*, 1993.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. 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*. North-Holland, 1992.

[IEE88] Institute of Electrical and Electronic Engineers, Inc., New York. *VHDL Language Reference Manual*, 1988. IEEE Standard 1076-1987.

[JS90] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.

[Lee92] Miriam E. Leeser. Using Nuprl for the verification and synthesis of hardware. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*. Prentice-Hall International, 1992.

[Men94a] Mentor-Graphics Corporation. *AutoLogic VHDL Synthesis Guide*, 1994.

[Men94b] Mentor-Graphics Corporation. *QuickVHDL Reference Manual*, 1994.

[MPT85] 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, *Computer Hardware Description Languages and Their Applications*. North-Holland, 1985.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Nav93] Zainalabedin Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill,Inc., 1993.

[OLHA95] John O'Leary, Miriam Leeser, Jason Hickey, and Mark Aagaard. Non-restoring integer square root: A case study in design by principled optimization. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume 901 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[OLLA93] John O'Leary, Mark Linderman, Miriam Leeser, and Mark Aagaard. HML: A hardware description language based on SML. In David Agnew, Luc Claesen, and Raul Camposano, editors, *Computer Hardware Description Languages and their Applications*, IFIP Transactions A-32, pages 327–334. Elsevier, North-Holland, 1993.

[OW94] Douglas E. Ott and Thomas J. Wilderotter. *A Designer's Guide to VHDL Synthesis*. Kluwer Academic Publishers, 1994.

[Pau91] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Rea89] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[TM91] Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.