Towards Provably-Correct Hardware Compilation Tools Based on Pass Separation Techniques

Steve McKeever and Wayne Luk

Department of Computing, Imperial College, 180 Queen's Gate, London, UK
{swm2,wl}@doc.ic.ac.uk

Abstract. This paper presents a framework for verifying compilation tools based on parametrised hardware libraries expressed in Pebble, a simple declarative language. An approach based on pass separation techniques is described for specifying and verifying Pebble abstraction mechanisms, such as the loop statement. We show how this approach can be used to verify the correctness of the flattening procedure in the Pebble compiler, which also results in a more efficient implementation than a non-verified version. The approach is useful for guiding compiler implementations for Pebble and related languages such as VHDL; it may also form the basis for automating the generation of provably-correct tools for hardware development.

1 Introduction

Advance in integrated circuit technology leads to an increasing emphasis on building designs from hardware libraries. A single parametrised library can be used to generate many implementations supporting multiple architectures, variable bit widths and trade-offs in speed and size. Such libraries enable effective hardware utilisation by exploiting technology-specific features whenever desirable, allowing designs with optimal performance and resource usage while minimising the need for knowing low-level details.

This paper describes an approach for developing provably-correct compilation tools for the Pebble language [12], which has been used to produce hardware libraries in VHDL, an industry standard language. While it is desirable to have hardware libraries in industry standard languages, there are, however, two major difficulties with developing VHDL libraries. First, VHDL is a versatile but complex language, and it takes much effort to write good parametrised code and to check its behaviour by simulation or other means – even when the subset used for realistic hardware libraries is small [13]. Second, most vendors have their own VHDL dialect; for instance not all VHDL tools support multi-dimensional vectors. It is unattractive to develop and maintain the same set of library elements in various vendor-specific dialects.

Our aim for Pebble is to enable application builders and library developers to work at a higher level of abstraction than that provided by VHDL, while ensuring that the resulting libraries are as flexible and efficient as those produced by hand. Our Pebble compiler targets various description formats, including parametrised and flattened VHDL and EDIF. It enables designers to compose and instantiate library elements, and it has been used to develop many designs for applications such as speech processing, data compression, and special video and graphics effects in augmented reality [14].

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 212–227, 2001.

An important component of the Pebble compiler is the flattening procedure, which produces flattened descriptions from hierarchical descriptions. Flattened descriptions are required by many tools, such as place-and-route programs for design implementation, and model checkers for design verification [18]. Interestingly, our proof of the flattening procedure not only offers users greater confidence in its correctness, it also leads to a more efficient implementation. Although this paper focuses on a specific proof, recent work indicates that similar techniques can be applied to verify other abstraction mechanisms, such as polymorphic variables and higher-order functions.

Pass separation [11] provides a framework for verifying abstraction mechanisms for generic descriptions, such as hierarchical blocks and `GENERATE-FOR` loops in Pebble, with respect to a Structural Operational Semantics [16]. For instance, the key to our proof is an environment invariant – Equation (1) – inspired by pass separation (Sect. 3.2). Such proofs are rare, but we feel that they are well-suited to verifying development tools for domain-specific languages containing multiple evaluation phases.

While much has been published on formal methods and tools for hardware design, it appears that most researchers focus on tools for producing correct designs [2],[18] rather than on the correctness of the tools themselves. Some researchers study embedding of synthesis algorithms in a theorem prover [4], or correctness condition generators for designs generated by synthesis tools [15]. Our work is complementary to their efforts, and is in a similar spirit to research on verifying compiler correctness for imperative descriptions for hardware [6],[7] and software [8] implementations.

2 Overview of Pebble

Pebble can be regarded as a much simplified variant of Structural VHDL. It provides a means of representing block diagrams hierarchically and parametrically [12]. A Pebble program is a block, defined by its name, parameters, interfaces, local definitions, and its body. The block interfaces are given by two lists, usually interpreted as the inputs and outputs. An input or an output can be of type `WIRE`, or it can be a multi-dimensional vector of wires. A wire can carry integer, boolean or other primitive data values.

A primitive block has an empty body; a composite block has a body containing the instantiation of composite or primitive blocks in any order. Blocks connected to each other share the same wire in the interface instantiation. For hardware designs, the primitive blocks can be bit-level logic gates and registers, or they can, like an adder, process word-level data such as integers or fixed-point numbers; the primitives depend on the availability of corresponding components in the domain targeted by the Pebble compiler. The `GENERATE-IF` statement enables conditional compilation and recursive definition, while the `GENERATE-FOR` statement allows the concise description of regular circuits.

Pebble has a simple, block-structured syntax. As an example, Fig. 2 describes the multiplexor array in Fig. 1, provided that the size parameter `n` is 4. In more complex descriptions, the parameters in a Pebble program can include the number of pipeline stages or the pitch between neighbouring interface connections [12]. Different network structures, such as tree- or butterfly-shaped circuits, can be described parametrically by indexing the components and wires.

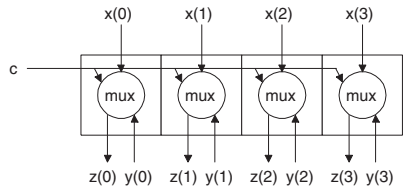

Fig. 1. An array of multiplexors described by the Pebble program in Fig. 2.

```
BLOCK muxarray (n:GENERIC)
        [c:WIRE, x,y:VECTOR (n-1..0) OF WIRE]
        [z:VECTOR (n-1..0) OF WIRE]
  VAR i
BEGIN
  GENERATE FOR i = 0..(n-1)
  BEGIN
    mux [c,x(i),y(i)] [z(i)]
  END
END;
```

Fig. 2. A description of an array of multiplexors (Fig. 1) in Pebble. The external input `c` is used to provide a common control input for each mutiplexor.

The semantics of Pebble depends on the behaviour of the primitive blocks and their composition in the target technology. Currently a synchronous circuit model is used in our tools (Sect. 3), and special control components for modelling run-time reconfiguration are also supported [12]. However, other models can be used if desired. Indeed Pebble can model any block-structured systems, not just electronic circuits.

Advanced features of Pebble include support for annotations and for modules. Such features improve design efficiency and reusability, and facilitate interface to components in other languages, including behavioural descriptions. Discussions about these features are beyond the scope of this paper.

3 Program Staging and Pass Separation

This section introduces a framework in which abstraction mechanisms for Pebble can be specified and verified. Our approach consists of three steps. The first is to provide a semantics for a flattened version of Pebble. The second is to characterise an abstraction mechanism in two ways: (a) specify how designs exploiting the abstraction mechanism can be transformed into flattened Pebble, and (b) provide the semantics of the abstraction mechanism directly. The third is to show that the semantics of a design produced by (a) is consistent with (b).

In the following, we specify and verify two Pebble abstraction mechanisms: hierarchical blocks, and GENERATE-FOR loops. The insight is to recognise that pass separation provides a framework for the above approach, so that the correctness of the two abstraction mechanisms can be demonstrated with respect to a Structural Operational Semantics [16] for Pebble. Only elementary understanding of such semantics is required to follow our work. We shall first present an overview of pass separation, and then explain how it can be used in verifying Pebble abstraction mechanisms. The key to our proof is an environment invariant – Equation (1) – inspired by pass separation.

Consider a repeated computation, part of whose input context remains invariant across all repetitions. Program staging is a technique which improves performance by separating the computation into two phases. We follow this approach to separate the task of interpreting a Pebble program on a variety of inputs: an early phase flattens the parametrised description into a collection of primitive gate calls, and a late phase completes the task given the varying inputs.

Two popular methods for separating a computation into stages are partial evaluation and pass separation [11]. We have used both methods in our study of Pebble and in tool development; in the following we shall focus on pass separation as a means to study abstraction mechanisms for Pebble.

Pass separation constructs, from a program $\mathtt{p}$, two programs $\mathtt{p}_1$, $\mathtt{p}_2$ such that:

$$[\![\mathtt{p}]\!]\ (\mathtt{x}, \mathtt{y}) = [\![\mathtt{p}_2]\!]\ ([\![\mathtt{p}_1]\!]\ \mathtt{x}, \mathtt{y})$$

for all $\mathtt{x}$ and $\mathtt{y}$, where $[\![\mathtt{p}]\!]$ is the function mapping the program $\mathtt{p}$ to its meaning. The equation indicates that $[\![\mathtt{p}]\!]$ can be split into two stages: computing $\mathtt{v} = [\![\mathtt{p}_1]\!]\,\mathtt{x}$ and $[\![\mathtt{p}_2]\!]\,(\mathtt{v}, \mathtt{y})$. The intention in performing pass separation is to "move" as many computations from $\mathtt{p}$ to $\mathtt{p}_1$ as possible, given only input $\mathtt{x}$. In our framework, let $\mathtt{p}$ be the semantics ($\mathcal{PS}$) of Pebble, $\mathtt{x}$ be a parametrised circuit description $\mathtt{C}$, and $\mathtt{y}$ be some input data. Then $\mathtt{p}_1$ corresponds to the abstraction mechanism, which in this case can be described using a flattening procedure ($\mathcal{FP}$). Similarly $\mathtt{p}_2$ corresponds to the semantics ($\mathcal{FS}$) of the flattened description on the data as shown in Fig. 3.

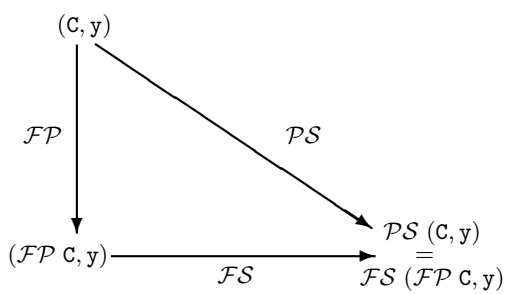

Fig. 3. Commuting diagram describing the pass separation equation.

We shall restrict our attention to a subset of Pebble which does not include vectors or GENERATE-IF statements. We begin by presenting the semantics of Flattened Pebble before adding the necessary structure to create Hierarchical Pebble. We then display a

procedure for instantiating the generics. We show that for a set of input values, a flattened description produces the same results as those from the hierarchical description.

3.1 Flattened Pebble

In its most basic form, a circuit consists of a collection of primitive block calls mapping input wires to output wires. Intermediate wires link these primitive block calls together. A circuit description is enclosed within a `main` block and primitive block identifiers are denoted by $id_{\mathcal{P}}$, as shown in the following syntax:

$$
\begin{array}{lll}
\mathit{circuit} & ::= & \texttt{BLOCK main} \\
 & & \quad [id_{in_1} : type_{in_1}, \ldots, id_{in_n} : type_{in_n}] \\
 & & \quad [id_{out_1} : type_{out_1}, \ldots, id_{out_m} : type_{out_m}] \\
 & & \quad dec_1 \texttt{;} \cdots \texttt{;} dec_j \\
 & & \texttt{BEGIN } stmts \texttt{ END} \\
type & ::= & \texttt{WIRE} \\
dec & ::= & \texttt{VAR } id \texttt{ : } type \\
stmts & ::= & stmt_1 \texttt{;} \cdots \texttt{;} stmt_k \\
stmt & ::= & id_{\mathcal{P}} \ [id_1, \ldots, id_n] \ [id_1, \ldots, id_m]
\end{array}
$$

Note that the language is applicative, as each wire is given an attribute only once. Input wires are defined and given the appropriate values; the purpose of the semantics is to find suitable definitions for the output wires. The semantic domain for wires is parametrised by the metavariable a, so that primitive objects can be of any type; it allows us to deal with both bit-level descriptions and word-level descriptions.

$$\textbf{data}\ \mathsf{wire\ a} = \mathsf{Undefined} \mid \mathsf{Defined\ a}$$

To deal with sequential circuits, the datatype a can be lifted to the stream domain [9].

The Structural Operational Semantics rules for Flattened Pebble, defined by $\rightarrow$ transitions [16], are given in Fig. 4. A local environment ρ maps identifiers to their wires. Primitive logic operators that map boolean pairs to booleans, such as xor, are held in an environment labeled δ. Such operators can only be applied to wires that are defined.

Two rules provide the meaning of primitive gate calls. If one or more inputs are Undefined, then the statement is returned unevaluated, as the gate call cannot be completed. The second rule applies the primitive function to the gate's parameters. Statements can be evaluated in any order; those that complete update the local environment ρ. When all statements have been reduced, the final environment is returned.

The output and intermediate wires of the `main` block are initially declared as Undefined. The block's statements are evaluated to calculate the final environment ρ', from which the output wires are extracted.

3.2 Hierarchical Pebble

Parametrised designs require the addition of a separate parameter list for generic values. Blocks other than `main` can receive values that define the bounds of loops or that can

$$\frac{\exists j \cdot 1 \leq j \leq n \wedge (\rho\ id_j) = \mathsf{Undefined}}{\delta \vdash \langle id_{\mathcal{P}}\ [id_1, \ldots, id_n]\ [id'_1, \ldots, id'_m], \rho\rangle \rightarrow_{stmt} \langle id_{\mathcal{P}}\ [id_1, \ldots, id_n]\ [id'_1, \ldots, id'_m], \rho\rangle}$$

$$\frac{(\rho\ id_1) = \mathsf{Defined}\ v_1\ \wedge\ \cdots\ \wedge\ (\rho\ id_n) = \mathsf{Defined}\ v_n \quad (\delta\ id_{\mathcal{P}})\ (v_1, \ldots, v_n) = (v'_1, \ldots, v'_m)}{\delta \vdash \langle id_{\mathcal{P}}\ [id_1, \ldots, id_n]\ [id'_1, \ldots, id'_m], \rho\rangle \rightarrow_{stmt} \rho \oplus \{id'_1 \mapsto \mathsf{Defined}\ v'_1, \cdots, id'_m \mapsto \mathsf{Defined}\ v'_m\}}$$

$$\frac{\delta \vdash \langle stmt_i, \rho\rangle \rightarrow_{stmt} \rho'}{\delta \vdash \langle stmt_1; \cdots stmt_{i-1}; stmt_i; stmt_{i+1}; \cdots stmt_k, \rho\rangle \rightarrow_{stmts} \langle stmt_1; \cdots stmt_{i-1}; stmt_{i+1}; \cdots stmt_k, \rho'\rangle}$$

$$\frac{\delta \vdash \langle stmt_i, \rho\rangle \rightarrow_{stmt} \langle stmt_i, \rho\rangle}{\delta \vdash \langle stmt_1; \cdots stmt_{i-1}; stmt_i; stmt_{i+1}; \cdots stmt_k, \rho\rangle \rightarrow_{stmts} \langle stmt_1; \cdots stmt_{i-1}; stmt_i; stmt_{i+1}; \cdots stmt_k, \rho\rangle}$$

$$\delta \vdash \langle [\,], \rho\rangle \rightarrow_{stmts} \rho$$

$$\frac{\begin{array}{l}\rho_1 = \{id_{in_1} \mapsto \mathsf{Defined}\ v_1, \cdots, id_{in_n} \mapsto \mathsf{Defined}\ v_n\} \\ \rho_2 = \{id_{out_1} \mapsto \mathsf{Undefined}, \cdots, id_{out_m} \mapsto \mathsf{Undefined}\} \\ \rho_3 = \{id_1 \mapsto \mathsf{Undefined}, \cdots, id_j \mapsto \mathsf{Undefined}\} \\ \rho = \rho_1 \oplus \rho_2 \oplus \rho_3 \\ \delta \vdash \langle stmts, \rho\rangle \rightarrow_{stmts} \rho'\end{array}}{\delta \vdash \left\langle \left(\begin{array}{l}\texttt{BLOCK main} \\ \texttt{[}id_{in_1}\texttt{:WIRE, ... ,}\ id_{in_n}\texttt{:WIRE]} \\ \texttt{[}id_{out_1}\texttt{:WIRE, ... ,}\ id_{out_m}\texttt{:WIRE]} \\ \quad \texttt{VAR}\ id_1\texttt{:WIRE; ... VAR}\ id_j\texttt{:WIRE} \\ \texttt{BEGIN}\ stmts\ \texttt{END}\end{array} \right), [v_1, \ldots, v_n] \right\rangle \rightarrow_{main} [(\rho'\ id_{out_1}), \ldots, (\rho'\ id_{out_m})]}$$

Fig. 4. Semantics of Flattened Pebble, based on $\rightarrow$ rules for *main*, *stmts* and *stmt*.

be passed to subsequent gate calls as defined in the syntax for Hierarchical Pebble:

$$\begin{array}{lll} circuit & ::= & main\texttt{;}\ block_1\texttt{;}\ \cdots\ \texttt{;}\ block_i \\ main & ::= & \texttt{BLOCK main} \\ & & \quad \texttt{[}id_{in_1}\texttt{:}type_{in_1}\texttt{, ... ,}\ id_{in_n}\texttt{:}type_{in_n}\texttt{]} \\ & & \quad \texttt{[}id_{out_1}\texttt{:}type_{out_1}\texttt{, ... ,}\ id_{out_m}\texttt{:}type_{out_m}\texttt{]} \\ & & dec_1\texttt{;}\ \cdots\ \texttt{;}\ dec_j \\ & & \texttt{BEGIN}\ stmts\ \texttt{END} \\ block & ::= & \texttt{BLOCK}\ id\ \texttt{(}id_{gen_1}\texttt{, ... ,}\ id_{gen_q}\texttt{)} \\ & & \quad \texttt{[}id_{in_1}\texttt{:}type_{in_1}\texttt{, ... ,}\ id_{in_n}\texttt{:}type_{in_n}\texttt{]} \\ & & \quad \texttt{[}id_{out_1}\texttt{:}type_{out_1}\texttt{, ... ,}\ id_{out_m}\texttt{:}type_{out_m}\texttt{]} \\ & & dec_1\texttt{;}\ \cdots\ \texttt{;}\ dec_j \\ & & \texttt{BEGIN}\ stmts\ \texttt{END} \end{array}$$

$$
\begin{array}{lll}
\mathit{type} & ::= & \texttt{WIRE} \\
\mathit{dec} & ::= & \texttt{VAR}\ \mathit{id}\ \texttt{:}\ \mathit{type} \mid \texttt{VAR}\ \mathit{id}\ \texttt{:}\ \texttt{NUM} \\
\mathit{stmts} & ::= & \mathit{stmt}_1\ \texttt{;}\ \cdots\ \texttt{;}\ \mathit{stmt}_k \\
\mathit{stmt} & ::= & \mathit{id}_{\mathcal{P}}\ [\mathit{id}_1, \ldots, \mathit{id}_n]\ [\mathit{id}_1, \ldots, \mathit{id}_m] \\
 & \mid & \mathit{id}\ (\mathit{exp}_1, \ldots, \mathit{exp}_q)\ [\mathit{id}_1, \ldots, \mathit{id}_n]\ [\mathit{id}_1, \ldots, \mathit{id}_m] \\
 & \mid & \texttt{GENERATE FOR}\ \mathit{id} = \mathit{exp}_1\ \texttt{..}\ \mathit{exp}_2 \\
 & & \texttt{BEGIN}\ \mathit{stmts}\ \texttt{END} \\
\mathit{exp} & ::= & \mathit{id} \mid n \mid \mathit{exp}_1\ \mathit{bop}\ \mathit{exp}_2 \mid \mathit{uop}\ \mathit{exp}
\end{array}
$$

The semantic rules for Hierarchical Pebble statements, defined by $\Rightarrow$ transitions, are given in Fig. 5. Two new environments are introduced: Γ maps block names to their bodies, while σ maps generic variables and loop indices to their values. Arithmetic expressions are evaluated by the valuation function $\mathcal{E}$ in an appropriate value environment. The rules for primitive gate calls and statement lists remain essentially unchanged except for the additional environments.

$$
\frac{\mathcal{E}_\sigma[\![\mathit{exp}_1]\!] > \mathcal{E}_\sigma[\![\mathit{exp}_2]\!]}{\Gamma, \delta, \sigma \vdash \left\langle \left(\begin{array}{l} \texttt{GENERATE FOR}\ \mathit{id}_{\mathit{index}} = \mathit{exp}_1\ \texttt{..}\ \mathit{exp}_2 \\ \texttt{BEGIN}\ \mathit{stmts}\ \texttt{END} \end{array} \right), \rho \right\rangle \Rightarrow_{\mathit{stmt}} \rho}
$$

$$
\frac{\begin{array}{c} \mathcal{E}_\sigma[\![\mathit{exp}_1]\!] \leq \mathcal{E}_\sigma[\![\mathit{exp}_2]\!] \\ \Gamma, \delta, \sigma \oplus \{\mathit{id} \mapsto \mathcal{E}_\sigma[\![\mathit{exp}_1]\!]\} \vdash \langle \mathit{stmts}, \rho \rangle \Rightarrow_{\mathit{stmts}} \rho' \\ \Gamma, \delta, \sigma \vdash \left\langle \left(\begin{array}{l} \texttt{GENERATE FOR}\ \mathit{id}_{\mathit{index}} = (\mathit{exp}_1\texttt{+1})\ \texttt{..}\ \mathit{exp}_2 \\ \texttt{BEGIN}\ \mathit{stmts}\ \texttt{END} \end{array} \right), \rho' \right\rangle \Rightarrow_{\mathit{stmt}} \rho'' \end{array}}{\Gamma, \delta, \sigma \vdash \left\langle \left(\begin{array}{l} \texttt{GENERATE FOR}\ \mathit{id}_{\mathit{index}} = \mathit{exp}_1\ \texttt{..}\ \mathit{exp}_2 \\ \texttt{BEGIN}\ \mathit{stmts}\ \texttt{END} \end{array} \right), \rho \right\rangle \Rightarrow_{\mathit{stmt}} \rho''}
$$

$$
\vdots
$$

$$
\frac{\begin{array}{l} (\Gamma\ \mathit{id}) = \left(\begin{array}{l} \texttt{BLOCK}\ \mathit{id}\ (\mathit{id}_{\mathit{gen}_1}, \ldots, \mathit{id}_{\mathit{gen}_q})\ [\mathit{id}_{\mathit{in}_1}\texttt{:WIRE}, \ldots, \mathit{id}_{\mathit{in}_n}\texttt{:WIRE}] \\ \qquad\qquad\qquad\qquad [\mathit{id}_{\mathit{out}_1}\texttt{:WIRE}, \ldots, \mathit{id}_{\mathit{out}_m}\texttt{:WIRE}] \\ \quad \texttt{VAR}\ \mathit{id}_{\mathit{index}_1}\ \texttt{:}\ \texttt{NUM;}\ \ldots\ \texttt{VAR}\ \mathit{id}_{\mathit{index}_j}\ \texttt{:}\ \texttt{NUM;} \\ \quad \texttt{VAR}\ \mathit{id}_{\mathit{local}_1}\ \texttt{:}\ \texttt{WIRE;}\ \ldots\ \texttt{VAR}\ \mathit{id}_{\mathit{local}_k}\ \texttt{:}\ \texttt{WIRE} \\ \texttt{BEGIN}\ \mathit{stmts}\ \texttt{END} \end{array} \right) \\ \quad \sigma_1 = \{\mathit{id}_{\mathit{gen}_1} \mapsto \mathcal{E}_\sigma[\![e_1]\!], \ldots, \mathit{id}_{\mathit{gen}_q} \mapsto \mathcal{E}_\sigma[\![e_q]\!]\} \\ \quad \rho_1 = \{\mathit{id}_{\mathit{in}_1} \mapsto (\rho\ \mathit{id}_1), \ldots, \mathit{id}_{\mathit{in}_n} \mapsto (\rho\ \mathit{id}_n)\} \\ \quad \rho_2 = \{\mathit{id}_{\mathit{out}_1} \mapsto \mathsf{Undefined}, \ldots, \mathit{id}_{\mathit{out}_m} \mapsto \mathsf{Undefined}\} \\ \quad \rho_3 = \{\mathit{id}_{\mathit{local}_1} \mapsto \mathsf{Undefined}, \ldots, \mathit{id}_{\mathit{local}_k} \mapsto \mathsf{Undefined}\} \\ \quad \rho' = \rho_1 \oplus \rho_2 \oplus \rho_3 \\ \Gamma, \delta, \sigma_1 \vdash \langle \mathit{stmts}, \rho' \rangle \Rightarrow_{\mathit{stmts}} \rho'' \end{array}}{\begin{array}{l} \Gamma, \delta, \sigma \vdash \langle \mathit{id}\ (e_1, \ldots, e_q)\ [\mathit{id}_1, \ldots, \mathit{id}_n]\ [\mathit{id}'_1, \ldots, \mathit{id}'_m], \rho \rangle \\ \qquad\qquad \Rightarrow_{\mathit{stmt}} \rho \oplus \{\mathit{id}'_1 \mapsto (\rho''\ \mathit{id}_{\mathit{out}_1}), \cdots, \mathit{id}'_m \mapsto (\rho''\ \mathit{id}_{\mathit{out}_m})\} \end{array}}
$$

Fig. 5. Semantics of Hierarchical Pebble. The $\Rightarrow$ rules dealing with primitive statements are similar to the corresponding $\rightarrow$ rules and are not shown.

Two rules are required for loops. The first rule deals with the situation when the loop has terminated. A loop has terminated when the expression representing the lower bound denotes a value that is greater than the upper bound. In such cases the current ρ environment is returned unchanged. Alternatively, the loop's statements are evaluated in a value environment where the loop index is bound to the lower bound value. Once completed, the loop is re-evaluated with the updated ρ' and a lower bound expression that has been incremented by one. The rule for enacting new block calls retrieves the block definition from the environment Γ; it creates a value environment σ_1 by mapping the generic variable names to their values calculated in the outer block's value environment σ; it creates an initial wire environment ρ' by mapping the input variable names to their wire values extracted from ρ and coalesced with Undefined bindings for output and intermediate variable names; and it evaluates the called block's statements to create a final environment ρ'', from which the output wires are extracted. The rule for the main block creates the initial environment ρ in much the same manner as with Flattened Pebble descriptions, and is shown in Fig. 6. The blocks statements are evaluated in an initial empty value environment.

A Hierarchical Pebble description can be flattened by unfolding both the generic variables and the `GENERATE-FOR` loops. The block environment Γ and the local variable environment σ support the abstraction mechanisms, and do not affect the underlying evaluation mechanism. Hence we can instantiate generic variables prior to the application of input wires, enabling block definitions to be flattened and incorporated into the `main` block. Flattened Pebble, itself a subset of Pebble, is used as the output language of this flattening process to facilitate its proof. To avoid parameters and local wire names overwriting each other when instantiating block calls, we rename all such variables beforehand using the function $\alpha :: block \rightarrow block$ (Fig. 7).

To model the static behaviour of the wire environment ρ in hierarchical descriptions with local bindings, we introduce a local environment μ which behaves like a symbol table mapping local parameter names to their original definitions, be they inputs to the circuit or local variable definitions. This leads to the invariant equation below, where ρ is the environment for modelling wire values in a hierarchical description, while ρ_d is a dynamic environment for flattened descriptions:

$$\forall id \cdot \rho\ id = \rho_d\ (\mu\ id) \tag{1}$$

This invariant equation will be used extensively in the correctness proof of the flattening procedure for Hierarchical Pebble in Sect. 4. The flattening procedure itself, defined by $\Downarrow$, is given in Fig. 7. Flattening a single statement will result in a pair of lists consisting of primitive gate calls and intermediate wire declarations. The statement list represents those primitive calls required to implement the statement derived from unfolding subsequent parametrised blocks, while the declarations are for the local wire definitions belonging to each unfolded block. The intention is to create the flattened `main` block from these two lists.

Primitive calls are simply returned with their parameter lists updated with their original variable definitions held in μ. Parametrised gate calls create a new instance of the retrieved block using the function α. Generic variables are bound to their values in σ_1. A static environment μ' is created by mapping the parameter names to their original

$$\frac{\begin{array}{c}\rho_1 = \{id_{in_1} \mapsto \mathsf{Defined}\ v_1, \cdots, id_{in_n} \mapsto \mathsf{Defined}\ v_n\} \\ \rho_2 = \{id_{out_1} \mapsto \mathsf{Undefined}, \cdots, id_{out_m} \mapsto \mathsf{Undefined}\} \\ \rho_3 = \{id_1 \mapsto \mathsf{Undefined}, \cdots, id_j \mapsto \mathsf{Undefined}\} \\ \Gamma, \delta, \{\} \vdash \langle stmts, \rho_1 \oplus \rho_2 \oplus \rho_3 \rangle \Rightarrow_{stmts} \ \rho'\end{array}}{\Gamma, \delta \vdash \left\langle \left(\begin{array}{l}\texttt{BLOCK main} \\ \texttt{[}id_{in_1}\texttt{:WIRE, ..., }id_{in_n}\texttt{:WIRE]} \\ \texttt{[}id_{out_1}\texttt{:WIRE, ..., }id_{out_m}\texttt{:WIRE]} \\ \quad \texttt{VAR }id_1\texttt{:WIRE; ... VAR }id_j\texttt{:WIRE} \\ \texttt{BEGIN }stmts\texttt{ END}\end{array} \right), [v_1, \ldots, v_n] \right\rangle \Rightarrow_{main} [(\rho'\ id_{out_1}), \ldots, (\rho'\ id_{out_m})]}$$

Fig. 6. Semantics of the `main` block in Pebble.

$$\Gamma, \mu, \sigma \vdash \langle id_{\mathcal{P}}\ [id_1, \ldots, id_n]\ [id'_1, \ldots, id'_m] \rangle \Downarrow_{stmt} ([id_{\mathcal{P}}\ [\mu\ id_1, \ldots, \mu\ id_n]\ [\mu\ id'_1, \ldots, \mu\ id'_m]], [\,])$$

$$(\alpha\ (\Gamma\ id)) = \left(\begin{array}{l}\texttt{BLOCK }id\ (id_{gen_1}, \ldots, id_{gen_q})\ \texttt{[}id_{in_1}\texttt{:WIRE, ..., }id_{in_n}\texttt{:WIRE]} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{[}id_{out_1}\texttt{:WIRE, ..., }id_{out_m}\texttt{:WIRE]} \\ \quad \texttt{VAR }id_{index_1}\texttt{ : NUM; ... VAR }id_{index_j}\texttt{ : NUM} \\ \quad \texttt{VAR }id_{local_1}\texttt{ : WIRE; ... VAR }id_{local_k}\texttt{ : WIRE} \\ \texttt{BEGIN }stmts\texttt{ END}\end{array} \right)$$

$$\frac{\begin{array}{c}\mu_1 = \{id_{in_1} \mapsto (\mu\ id_1), \ldots, id_{in_n} \mapsto (\mu\ id_n)\} \\ \mu_2 = \{id_{out_1} \mapsto (\mu\ id'_1), \ldots, id_{out_m} \mapsto (\mu\ id'_m)\} \\ \mu_3 = \{id_{local_1} \mapsto id_{local_1}, \ldots, id_{local_k} \mapsto id_{local_k}\} \\ \sigma_1 = \{id_{gen_1} \mapsto \mathcal{E}_\sigma[\![e_1]\!], \ldots, id_{gen_q} \mapsto \mathcal{E}_\sigma[\![e_q]\!]\} \\ \Gamma, \mu_1 \oplus \mu_2 \oplus \mu_3, \sigma_1 \vdash \langle stmts \rangle \Downarrow_{stmts} (stmts', locals')\end{array}}{\begin{array}{l}\Gamma, \mu, \sigma \vdash \langle id\ (e_1, \ldots, e_q)\ [id_1, \ldots, id_n]\ [id'_1, \ldots, id'_m] \rangle \\ \qquad \Downarrow_{stmt} (stmts', [\texttt{VAR }id_{local_1}\texttt{ : WIRE; } \cdots \texttt{; VAR }id_{local_k}\texttt{ : WIRE}] +\!\!+ locals')\end{array}}$$

$$\frac{\mathcal{E}_\sigma[\![exp_1]\!] > \mathcal{E}_\sigma[\![exp_2]\!]}{\Gamma, \mu, \sigma \vdash \left\langle \left(\texttt{GENERATE FOR }id_{index}\texttt{=}exp_1\texttt{..}\ exp_2\ \texttt{BEGIN }stmts\ \texttt{END} \right) \right\rangle \Downarrow_{stmt} ([\,], [\,])}$$

$$\frac{\begin{array}{c}\mathcal{E}_\sigma[\![exp_1]\!] \leq \mathcal{E}_\sigma[\![exp_2]\!] \\ \Gamma, \mu, \sigma \oplus \{id_{index} \mapsto \mathcal{E}_\sigma[\![exp_1]\!]\} \vdash \langle stmts \rangle \Downarrow_{stmts} (stmts', locals') \\ \Gamma, \mu, \sigma \vdash \left\langle \left(\begin{array}{l}\texttt{GENERATE FOR }id_{index}\texttt{=(}exp_1\texttt{+1)..}\ exp_2 \\ \texttt{BEGIN }stmts\texttt{ END}\end{array} \right) \right\rangle \Downarrow_{stmt} (stmts'', locals'')\end{array}}{\begin{array}{c}\Gamma, \mu, \sigma \vdash \left\langle \left(\begin{array}{l}\texttt{GENERATE FOR }id_{index}\texttt{=}exp_1\texttt{..}\ exp_2 \\ \texttt{BEGIN }stmts\texttt{ END}\end{array} \right) \right\rangle \\ \Downarrow_{stmt} (stmts' +\!\!+ stmts'', locals' +\!\!+ locals'')\end{array}}$$

$$\frac{\Gamma, \mu, \sigma \vdash \langle stmt_1 \rangle \Downarrow_{stmt} (stmts_1, locals_1) \cdots \Gamma, \mu, \sigma \vdash \langle stmt_n \rangle \Downarrow_{stmt} (stmts_n, locals_n)}{\Gamma, \mu, \sigma \vdash \langle stmt_1\texttt{; } \cdots \texttt{; }stmt_n \rangle \Downarrow_{stmts} (stmts_1 +\!\!+ \cdots +\!\!+ stmts_n, locals_1 +\!\!+ \cdots +\!\!+ locals_n)}$$

Fig. 7. Transition rules for flattening Pebble statements, based on $\Downarrow$ rules for *stmts* and *stmt*.

names held in μ. Local variables are bound to themselves, but are returned as declarations so that they are properly declared at run time. The blocks statements are flattened and returned with the local wire declarations. The two rules for loops apply the unfolding procedure at compile time to their subterms by incrementing the loop bound, creating primitive gate calls to implement the loop at run time. The rule for statements flattens each statement, and collects the intermediate calls and declarations together.

The flattening rule for the main block is shown in Fig. 8. An initial static environment μ is created binding input, output and local wire names to themselves as these will be their run-time names. The body of the block is flattened with an empty value environment. The returned list of primitives gate calls forms the body of the flattened main block and the derived intermediate wires are declared local to this block.

$$\frac{\begin{array}{c}\mu_1 = \{id_{in_1} \mapsto id_{in_1}, \cdots, id_{in_n} \mapsto id_{in_n}\} \\ \mu_2 = \{id_{out_1} \mapsto id_{out_1}, \cdots, id_{out_m} \mapsto id_{out_m}\} \\ \mu_3 = \{id_1 \mapsto id_1, \cdots, id_j \mapsto id_j\} \\ \mu = \mu_1 \oplus \mu_2 \oplus \mu_3 \\ \Gamma, \mu, \{\} \vdash \langle stmts \rangle \Downarrow_{stmts} (stmts', [\texttt{VAR } id'_1\texttt{:WIRE; ... VAR } id'_k\texttt{:WIRE}])\end{array}}{\Gamma \vdash \left\langle \begin{pmatrix}\texttt{BLOCK main } [id_{in_1}\texttt{:WIRE, ..., } id_{in_n}\texttt{:WIRE}] \\ \qquad\qquad [id_{out_1}\texttt{:WIRE, ..., } id_{out_m}\texttt{:WIRE}] \\ \texttt{VAR } id_1\texttt{:WIRE; ... VAR } id_j\texttt{:WIRE} \\ \texttt{BEGIN } stmts \texttt{ END}\end{pmatrix} \right\rangle \Downarrow_{main} \begin{pmatrix}\texttt{BLOCK main } [id_{in_1}\texttt{:WIRE, ..., } id_{in_n}\texttt{:WIRE}] \\ \qquad\qquad [id_{out_1}\texttt{:WIRE, ..., } id_{out_m}\texttt{:WIRE}] \\ \texttt{VAR } id_1\texttt{:WIRE; ... VAR } id_j\texttt{:WIRE;} \\ \texttt{VAR } id'_1\texttt{:WIRE; ... VAR } id'_k\texttt{:WIRE} \\ \texttt{BEGIN } stmts' \texttt{ END}\end{pmatrix}}$$

Fig. 8. Transition rules for flattening the `main` block.

To illustrate how blocks are flattened and local variables are renamed, we shall use the following example which creates a row of two not-gates:

```
BLOCK notrow (n) [vin:WIRE] [vout:WIRE]
  VAR inter:VECTOR (n..0) OF WIRE
  VAR i:NUM
BEGIN
  connect [vin] [inter(0)];
  GENERATE FOR i=0..n-1
  BEGIN
     not [inter(i)] [inter(i+1)]
  END;
  connect [inter(n)] [vout]
END;
```

```
BLOCK main [x:WIRE] [y:WIRE]
BEGIN
  notrow (2) [x] [y]
END;
```

The circuit is defined in terms of two primitive components: `connect` which links two wires together, and the `not` gate. Figure 9 demonstrates how a block call is flattened given previously established environments. The called block is initially renamed; the environment σ is created for the generic values; the environment μ maps parameter names to their original names; the block's statements are then flattened to create a list of primitive calls, and returned along with the distinct local wire definitions.

$$\frac{\vdots}{\Gamma, \mu', \sigma_1 \vdash \left\langle \begin{array}{l} \texttt{connect [vin1] [i1(0)];} \\ \texttt{GENERATE FOR } \cdots \\ \quad \vdots \\ \texttt{connect [i1(n1)] [vot1]} \end{array} \right\rangle \Downarrow_{stmts} (\texttt{[connect [x] [i1(0)];} \; \texttt{not [i1(0)] [i1(1)];} \; \texttt{not [i1(1)] [i1(2)];} \; \texttt{connect [i1(2)] [y]],} \; \texttt{[]})}$$

$\sigma_1 = \{\texttt{n1} \mapsto 2\}$
$\mu_1 = \{\texttt{vin1} \mapsto \texttt{x}\} \quad \mu_2 = \{\texttt{vot1} \mapsto \texttt{y}\} \quad \mu_3 = \{\texttt{i1} \mapsto \texttt{i1}\}$
$\mu' = \mu_1 \oplus \mu_2 \oplus \mu_3$

$(\alpha(\Gamma\ \texttt{notrow})) =$

```
BLOCK notrow (n1) [vin1:WIRE] [vot1:WIRE]
  VAR i1:VECTOR (n1..0) OF WIRE;
  VAR i2:NUM
BEGIN
  connect [vin1] [i1(0)];
  GENERATE FOR i2=0..n1-1
  BEGIN  not [i1(i2)] [i1(i2+1)]  END;
  connect [i1(n1)] [vot1]
END;
```

$$\Gamma, \{\texttt{x} \mapsto \texttt{x}, \texttt{y} \mapsto \texttt{y}\}, \{\} \vdash \langle \texttt{notrow (2) [x] [y]} \rangle \Downarrow_{stmt} (\texttt{[connect [x] [i1(0)];} \; \texttt{not [i1(0)] [i1(1)];} \; \texttt{not [i1(1)] [i1(2)];} \; \texttt{connect [i1(2)] [y]],[]}) \; \texttt{[VAR i1:VECTOR (2..0) OF WIRE]})$$

Fig. 9. Fragment of the proof tree for flattening `notrow`.

4 Correctness Proof

We can now present the main correctness theorem for flattening hierarchical blocks and `GENERATE-FOR` loops. This result, given by Equation (2), relies on Lemmas (3), (4) and (5). Each lemma is an instance of the commuting diagram given in Fig. 3, and involves the environment invariant given by Equation (1). At each syntactic level, they show how the sequence of outputs generated by the hierarchical definition can be calculated by first flattening the term and then using the simplified semantics. With the addition of `GENERATE-IF` statements, recursive block definitions can result in non-terminating programs. In these cases the flattening procedure will also fail to terminate.

The main theorem states that, from a given Hierarchical Pebble description consisting of a `main` block, a block environment Γ, a primitive gate environment δ, and a sequence of input wires $[v_1, \ldots, v_n]$, one can calculate the sequence of outputs derived from the circuit's proof trees by first unfolding the description to create one large `main` block, to which the flattening rules can be applied:

$$\begin{array}{l} \Gamma, \delta \vdash \langle main, [v_1, \ldots, v_n] \rangle \Rightarrow_{main} [v'_1, \ldots, v'_m] \\ \quad \Longrightarrow \Gamma \vdash main \Downarrow_{main} main' \\ \quad \wedge \quad \delta \vdash \langle main', [v_1, \ldots, v_n] \rangle \rightarrow_{main} [v'_1, \ldots, v'_m] \end{array} \tag{2}$$

This result can be proved by structural induction on the rules for the main block (Figs. 4, 6 and 8). We establish the invariant on wire environments initially, and we use Lemma (3) below to show how it holds on completion of the block's statements so that the same final values are derived.

From a given Hierarchical Pebble statement list, a block environment Γ, a primitive gate environment δ, a local value environment σ and a local wire environment ρ, we can calculate the wire bindings ρ' derived from a successful completion of the statements, by staging the computation in two. The first stage flattens the statements into a list of primitive calls, where local wire names are mapped to their original definitions using the static environment μ, and a distinct list of local wire declarations is created. The second stage applies the Flattened Pebble rules to the primitive gate call list using the dynamic wire environment ρ_d. An environment ρ'_d can be derived that will contain the same bindings as those for the hierarchical statements. This implication requires the invariant, given by Equation (1), to hold for wire environments:

$$\begin{array}{l} \Gamma, \delta, \sigma \vdash \langle stmts, \rho \rangle \Rightarrow_{stmts} \rho' \\ \quad \Longrightarrow \Gamma, \sigma, \mu \vdash stmts \Downarrow_{stmts} (stmts', locals') \\ \quad \wedge \quad \forall id \cdot \rho\ id = \rho_d(\mu\ id) \\ \quad \wedge \quad \forall id \cdot \rho'\ id = \rho'_d(\mu\ id) \\ \quad \wedge \quad \delta \vdash \langle stmts', \rho_d \rangle \rightarrow_{stmts} \rho'_d \end{array} \tag{3}$$

This lemma can be proved by induction on the length of derivation sequences using Lemma (4) below; it completes the presentation of the main theorem.

The next lemma deals mainly with `GENERATE-FOR` loops (Figs. 4, 5 and 7). From a given Hierarchical Pebble statement, a block environment Γ, a primitive gate environment δ, a local value environment σ, and a local wire environment ρ, one can calculate a set of wire bindings ρ' derived from the successful completion of the statement, by first

flattening the statement using the static environment μ, and then executing the derived statements in the dynamic wire environment ρ_d:

$$\begin{aligned} \Gamma, \delta, \sigma \ &\vdash\ \langle stmt, \rho\rangle \Rightarrow_{stmt} \rho' \\ &\Longrightarrow \Gamma, \sigma, \mu \ \vdash\ stmt \Downarrow_{stmt} (stmts', locals') \\ &\wedge \quad \forall id \cdot \rho\ id = \rho_d(\mu\ id) \\ &\wedge \quad \forall id \cdot \rho'\ id = \rho'_d(\mu\ id) \\ &\wedge \quad \delta \ \vdash\ \langle stmts', \rho_d\rangle \rightarrow_{stmt} \rho'_d \end{aligned} \tag{4}$$

This result can be proved by structural induction on statements: primitive gate calls, parametrised gate calls and loops. The first two cases are straightforward, once the invariants of the environments have been established. The third case, however, requires Lemma (5) to show that the appropriate final environment can be derived after staging:

$$\begin{aligned} &\left(\begin{array}{r} \Gamma, \delta, \sigma_1 \ \vdash\ \langle stmts_1, \rho\rangle \Rightarrow_{stmts} \rho' \\ \wedge\ \Gamma, \delta, \sigma_2 \ \vdash\ \langle stmts_2, \rho'\rangle \Rightarrow_{stmts} \rho'' \end{array}\right) \\ &\qquad \Longrightarrow \Gamma, \sigma_1, \mu \ \vdash\ stmts_1 \Downarrow_{stmts} (stmts'_1, locals'_1) \\ &\qquad \wedge \quad \Gamma, \sigma_2, \mu \ \vdash\ stmts_2 \Downarrow_{stmts} (stmts'_2, locals'_2) \\ &\qquad \wedge \quad \forall id \cdot \rho\ id = \rho_d(\mu\ id) \\ &\qquad \wedge \quad \forall id \cdot \rho''\ id = \rho''_d(\mu\ id) \\ &\qquad \wedge \quad \delta \ \vdash\ \langle stmts_1 +\!\!+ stmts_2, \rho_d\rangle \rightarrow_{stmts} \rho''_d \end{aligned} \tag{5}$$

This lemma states that reducing a statement list $stmts_1$ in the value environment σ_1 with wire bindings ρ, followed by reducing a second statement list $stmts_2$ in σ_2 yielding a final wire environment ρ'', can be derived by first flattening the two statement lists and then executing the concatenation of the two primitive gate call lists. The lemma can be proved by induction on the length of derivation sequences.

5 Compiler Development

This section reflects on the implications of our approach for compiler development. Natural semantic rules, as used in specifying Pebble, rely on notions of pattern matching, inference rules and operational semantics. They can be captured in a theorem prover [2], [17] or translated into Horn clauses via a metalanguage such as Typol [3]. Since the transition rules for flattening Hierarchical Pebble descriptions permit a left-right and top-down construction of the proof tree with no backtracking, we can replace a resolution engine by a functional evaluator based on pattern matching [1] to improve efficiency.

In practice, an implementation in a functional language of the core flattening procedure (Fig. 10) follows naturally from the rules in Fig. 7. For a particular language construct, a function definition is created that pattern matches its goal and obtains the result from the intermediate transitions by means of a `where` clause. This technique offers a way of automatically producing a functional implementation of the compilation tools directly from their specifications.

It is educational to compare the original, hand-developed implementation of the flattening procedure, and the new version in Fig. 10 which results from the specification and proof exercises. The new version is better than the original version in all aspects:

```
data Exp  = Number Int | Var String
          | Binop (Exp,Bop,Exp) | Unop (Uop,Exp)
data Type = WIRE
data Dec  = VARW (String,Type) | VARN String
data Stmt = PrimCall (String,[String],[String])
          | BlkCall (String,[Exp],[String],[String])
          | Loop (String,Exp,Exp,[Stmt])
data Blk  = Block (String,[String],
            [(String,Type)],[(String,Type)],[Dec],[Stmt])

fetch :: [(String,a)] -> String -> a
...
eval_exp :: Exp -> [(String,Int)] -> Int
...

flatten_stmt :: ([(String,Blk)],
  [(String,String)],[(String,Int)]) -> Stmt -> ([Stmt],[Dec])
flatten_stmt (gamma,mu,sigma) (PrimCall (pnm,args1,args2))
  =  ([PrimCall (pnm,map (fetch mu) args1,
                     map (fetch mu) args2)],[])
flatten_stmt (gamma,mu,sigma) (BlkCall (bnm,gens,args1,args2))
  =  (stmts',[(VARW d) | (VARW d) <- decs] ++ locals')
     where
          (Block (nm,gennms,parms1,parms2,decs,stmts))
                 = rename (fetch gamma bnm)
          sigma1 = zip gennms
                   (map (\ e -> eval_exp e sigma) gens)
          mu1    = zip parms1 (map (fetch mu) args1)
          mu2    = zip parms2 (map (fecth mu) args2)
          mu3    = [(id,id) | (VARW (id,WIRE)) <- decs]
          mu'    = mu1 ++ mu2 ++ mu3
          (stmts',locals')
                 = flatten_stmts (gamma,mu',sigma1) stmts
flatten_stmt (gamma,mu,sigma) (Loop (nm,e1,e2,stmts))
   | n1>n2       = ([],[])
   | otherwise   = (stmts'++stmts'',locals'++locals'')
     where
       n1 = eval_exp e1 sigma
       n2 = eval_exp e2 sigma
       (stmts',locals')
          = flatten_stmts (gamma,mu,(nm,n1):sigma) stmts
       (stmts'',locals'')
          = flatten_stmt (gamma,mu,sigma)
                  (Loop (nm,Binop (e1,Add,Number 1),e2,stmts))
flatten_stmts :: ([(String,Blk)],
  [(String,String)],[(String,Int)]) -> [Stmt]-> ([Stmt],[Dec])
flatten_stmts (gamma,mu,sigma) = unzip . map flatten_stmt
```

Fig. 10. Flattening Pebble in Haskell.

it is clearer, more concise, more robust and more efficient. The main reason is that, based on the formal development, the new version separates variable renaming from the flattening procedure. The original implementation, in contrast, attempted the former without properly considering the latter: the mingling of the two leads to situations where the renaming process is deeply nested within the unfolding procedure, leaving little scope for further optimisations. The new version is amenable to further optimisations, such as the use of de Bruijn indices to avoid the costly rename function [5]. Further refinements would lead to a highly efficient imperative implementation.

Our experience shows that deriving provably-correct compiler implementations can benefit their efficiency, in addition to increasing the confidence in their correctness. Furthermore, it appears possible that the verification of such implementations for domain-specific languages such as Pebble can be mechanised using custom proof engines [2]. Further work, however, is needed to explore this possibility thoroughly.

6 Concluding Remarks

While the version of Pebble described in this paper does not include advanced abstraction mechanisms, current work involves extending Pebble with polymorphic variables, records and higher-order functions. These features enable a combinator style of development [10] that tends to simplify the hardware design process.

Our extended compilation strategy infers the types of the polymorphic variables, unfolds the record definitions and instantiates higher-order functions prior to compile time to create a Hierarchical Pebble description. The correctness proof for Polymorphic Pebble is very similar to that for Hierarchical Pebble. An intermediate environment mapping polymorphic variables to types is used to create distinct blocks, and it leads to an invariant equation similar to Equation (1). Higher-Order Pebble enables nested function calls which require lambda lifting before the calls can be unfolded. In this way the ability to generate correct parametrised VHDL will be maintained.

The combinator style of description facilitates the formulation of correctness-preserving algebraic transformations for design development [10]. The proposed extensions of Pebble take us a step closer to providing, for instance, a generic transformation rule [13] which can be used to derive pipelined designs from a non-pipelined design. Further work will generalise our approach to deal with relational descriptions [10].

Our research contributes to insights about abstraction mechanisms and their validated implementations. It also provides a useful foundation on which further work, such as verifying tools for pipeline optimisation [19], can be based. We believe that provably-correct tools will have a profound impact on understanding the scope and effectiveness of hardware synthesis algorithms and their implementation.

Acknowledgements. Our thanks to the many Pebble developers, users and advisors, particularly Chris Booth, Ties Bos, Arran Derbyshire, Florent Dupont-De-Dinechin, Mike Gordon, He Jifeng, Tony Hoare, Danny Lee, James Rice, Richard Sandiford, Seng Shay Ping, Nabeel Shirazi, Dimitris Siganos, Henry Styles, Tim Todman and Markus Weinhardt, for their patience, help and encouragement. Thanks also to the comments and suggestions of anonymous reviewers of this paper. We acknowledge the support of

UK Engineering and Physical Sciences Research Council, Celoxica, and Xilinx. This work was carried out as part of Technology Group 10 of UK MOD's Corporate Research Programme.

References

1. I. Attali, J. Chazarain and S. Gilette, "Incremental evaluation of natural semantics specifications", *Proc. 4th Int. Symp. on Programming Language Implementation and Logic Programming*, LNCS 631, Springer, 1992.
2. L.A. Dennis et. al., "The PROSPER toolkit", *Proc. 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1785, Springer, 2000.
3. T. Despeyroux, "Executable specification of static semantics", *Semantics of Data Types*, LNCS 173, Springer, 1984.
4. D. Eisenbiegler, C. Blumenroehr and R. Kumar, "Implementation issues about the embedding of existing high level synthesis algorithms in HOL", *Theorem Proving in Higher Order Logics*, LNCS 1125, Springer, 1996.
5. C. Hankin, *Lambda Calculus, A guide for Computer Scientists*, Oxford University Press, 1994.
6. J. He, G. Brown, W. Luk and J.W. O'Leary, "Deriving two-phase modules for a multi-target hardware compiler", *Designing Correct Circuits*, Springer Electronic Workshop in Computing, 1996.
7. J. He, I. Page and J.P. Bowen, "Towards a provably correct hardware implementation of occam", *Correct Hardware Design and Verification Methods*, LNCS 683, Springer, 1993.
8. C.A.R. Hoare, J. He and A. Sampaio, "Normal form approach to compiler design", *Acta Informatica*, Vol. 30, pp. 701–739, 1993.
9. G. Jones and M. Sheeran, "Timeless truths about sequential circuits", *Concurrent Computations: Algorithms, Architectures and Technology*, S.K. Tewksbury et. al. (eds.), Plenum Press, 1988.
10. G. Jones and M. Sheeran, "Circuit design in Ruby", *Formal Methods for VLSI Design*, J. Staunstrup (ed.), North-Holland, 1990.
11. U. Jørring and W. Scherlis, "Compilers and staging transformations", *Proc. ACM Symp. on Principles of Programming Languages*, ACM Press, 1986.
12. W. Luk and S.W. McKeever, "Pebble: a language for parametrised and reconfigurable hardware design", *Field-Programmable Logic and Applications*, LNCS 1482, Springer, 1998.
13. W. Luk et. al., "A framework for developing parametrised FPGA libraries", *Field-Programmable Logic and Applications*, LNCS 1142, Springer, 1996.
14. W. Luk et. al., "Reconfigurable computing for augmented reality", *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 1999.
15. N. Mansouri and R. Vemuri, "A methodology for completely automated verification of synthesized RTL designs and its integration with a high-level synthesis tool", *Formal Methods in Computer-Aided Design*, LNCS 1522, Springer, 1998.
16. H.R. Nielson and F. Nielson, *Semantics with Applications*, John Wiley and Sons, 1992.
17. F. Pfenning and C. Schrmann, "System description: Twelf – a meta-logical framework for deductive systems", *Proc. Int. Conf. on Automated Deduction*, LNAI 1632, Springer, 1999.
18. M. Sheeran, S. Singh and G. Stalmarck, "Checking safety properties using induction and a SAT-solver", *Proc. Int. Conf. on Formal Methods in CAD*, LNCS 1954, Springer, 2000.
19. M. Weinhardt and W. Luk, "Pipeline vectorization", *IEEE Trans. CAD*, Vol. 20, No. 2, 2001, pp. 234–248.