

Provably-correct hardware compilation tools based on pass separation techniques

Steve McKeever¹ and Wayne Luk²

¹ Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
E-mail: Steve.McKeever@comlab.ox.ac.uk

² Department of Computing, Imperial College, 180 Queen's gate, London, UK
E-mail: wl@doc.ic.ac.uk

Abstract. This paper presents a framework for verifying compilation tools for parametrised hardware designs with placement information. The framework involves Pebble, a simple declarative language based on Structural VHDL which supports the use of placement information to guide circuit layout; such information often leads to efficient designs that are particularly important for hardware libraries. Relative placement information enables control of circuit layout at a higher level of abstraction than placement information in the form of explicit coordinates. An approach based on pass separation techniques is adopted for specifying and verifying two Pebble abstraction mechanisms: a flattening procedure and a relative placement method. For the flattening procedure, which takes a set of parametrised blocks and unfolds the circuit description into a netlist, we provide semantic descriptions of both the hierarchical and the flattened Pebble languages to prove its functional correctness. For the relative placement method, we specify the compilation procedure from Pebble programs with relative placement information to Pebble programs with explicit coordinate expressions, often in the form of symbolic placement constraints. This compilation procedure can be used in conjunction with partial evaluation to optimise the size and speed of parametrised circuit descriptions using relative placement, without flattening the original hierarchical descriptions. Our approach has been used for optimising a pattern matcher design, which results in a 33% reduction in resource usage. For DES encryption, our method can reduce the size of a DES design by 60%.

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 [LuM97], 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 [Luk96]. 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 research involving Pebble has two aspects. The first 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 data encryption [MLD02] and special video and graphics effects in augmented reality [Luk99].

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 partitioning tools for physical design [Con01], and model checkers for design verification [SSS00]. Interestingly, our proof of the flattening procedure not only offers users greater confidence in its correctness, it also leads to a more efficient implementation.

The second aspect of our research is in supporting relative placement. Relative placement information enables control of circuit layout at a higher level of abstraction than placement information in the form of explicit coordinates. It has been shown that, despite advances in automatic placement methods, user-supplied placement information can often significantly improve FPGA performance and resource utilization for common applications [S00]. Relative placement for hardware design has been developed for languages such as μ FP [LJS89], Ruby [GL01] and Lava [S00]. However, all hardware languages that support such placement techniques are compiled into a netlist in a single stage. This method makes it difficult to interface libraries written using explicit coordinates with those designed using relative placement information. It is therefore important that placement information, which guides design tools to produce efficient designs, should be supported in hardware library descriptions. Our method uses the notions of *Beside* and *Below* to describe relative placement of circuit cells and blocks. Our compilation technique projects such blocks onto a coordinate grid such that no overlapping can occur.

Pass separation [JøS86] provides a framework for correctness proof of both abstraction mechanisms. Such proofs are rare, but we feel that they are well-suited to verifying development tools for domain-specific languages containing multiple evaluation phases. These phases can be effectively combined so that compilation becomes a process of refinement from an abstract description to a more concrete version.

The result from pass separation can be expressed in a language different from the one used in describing hardware designs. For example, our compiler converts hardware descriptions with relative placement information to those with explicit placement information with a different syntax. In contrast, the result from partial evaluation [JGS93] is usually expressed in a language which is a subset of the original.

To summarise, the contributions of this paper are:

1. A framework based on pass separation techniques for specifying, verifying and implementing a flattening procedure for hardware designs that are hierarchical and parametrised (Sects. 3, 4 and 5),
2. Extension of this framework for compiling and verifying designs with relative placement information (Sects. 6, 7 and 8),
3. Further extension of this framework to support conditionals, which forms the basis of a partial evaluation method for design compaction (Sect. 9).

While much has been published on formal methods and tools for hardware design [KeG99], it appears that most researchers focus on tools for producing correct designs [Den00, SSS00] rather than on the correctness of the tools themselves [KBE96]. Other relevant work includes embedding high-level synthesis algorithms in HOL [EBK96], and integrating automated verification of synthesised designs with a high-level synthesis tool [Mav98]. An example of a verified tool is PBS, a multi-level logic synthesis program based on the weak division algorithm which has been verified using the Nuprl proof development system [AaL94, AaL95]. Another example is a run-time specialisation algorithm for the Xilinx XC6200 devices, which has been verified by the PVS prover [SM98].

Our research is complementary to their efforts: it focuses on a framework for specifying and verifying hardware abstraction mechanisms, rather than on formalising and mechanising the proof of a particular logic synthesis algorithm. It is in a similar spirit to research on verifying compiler correctness for imperative descriptions for hardware [HBL96, HPB93] and software [HHA93] implementations. Currently our proofs have been undertaken by hand. While we are confident in their correctness, it would be useful and interesting to explore both their mechanisation and their recasting using `fold` and `unfold` operators [Hut98].

The rest of this paper is organised as follows. Section 2 provides an overview of the Pebble language. Section 3 describes the flattening procedure for Pebble, while Sect. 4 outlines its proof and Sect. 5 provides its implemen-

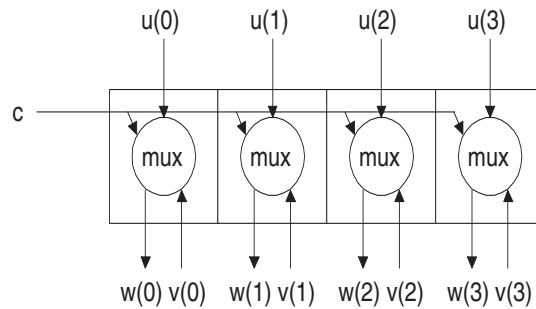


Fig. 1. An array of multiplexers described by the Pebble program in Fig. 2

tation. Section 6 extends Pebble with relative placement information and illustrates its features using a simple pattern matcher. Section 7 describes how the relative placement information is projected onto a coordinate scheme using arithmetic expressions with symbolic constraints. Section 8 outlines the proof of this procedure. Section 9 shows how to integrate the relative placement information with conditionals, and discusses how this approach enables compaction by partial evaluation. Section 10 provides a summary and describes on-going research.

2. Overview of Pebble

The Pebble language can be regarded as a much simplified variant of Structural VHDL. It provides a means of representing block diagrams hierarchically and parametrically [LuM97]. 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, the multiplexer array in Fig. 1, is described in Fig. 2, 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 [LuM97]. Different network structures, such as tree- or butterfly-shaped circuits, can be described parametrically by indexing the components and wires.

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 [LuM97]. 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 step is to provide a semantics for a flattened version of Pebble. The second step 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

```

BLOCK muxarray (n:GENERIC)
  [c:WIRE, u,v:VECTOR (n-1..0) OF WIRE]
  [w:VECTOR (n-1..0) OF WIRE]
  VAR i
  BEGIN
    GENERATE FOR i = 0..(n-1)
    BEGIN
      mux [c,u(i),v(i)] [w(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 multiplexor

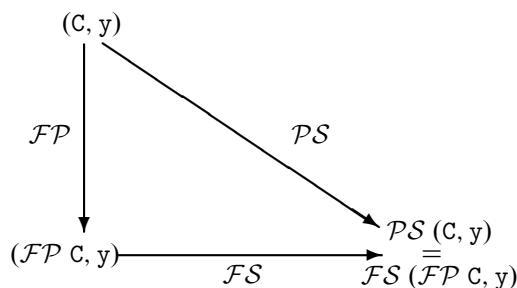


Fig. 3. Commuting diagram describing the pass separation equation

abstraction mechanism directly. The third step 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 [NiN92] 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 (Eq. 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 [JøS86]. 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; we shall illustrate the use of partial evaluation in Sect. 9.

Pass separation constructs, from a program p , two programs p_1, p_2 such that:

$$\llbracket p \rrbracket (x, y) = \llbracket p_2 \rrbracket (\llbracket p_1 \rrbracket x, y)$$

for all x and y , where $\llbracket p \rrbracket$ is the function mapping the program p to its meaning. The equation indicates that $\llbracket p \rrbracket$ can be split into two stages: computing $v = \llbracket p_1 \rrbracket x$ and $\llbracket p_2 \rrbracket (v, y)$. The intention in performing pass separation is to “move” as many computations from p to p_1 as possible, given only input x . In our framework, let p be the semantics (PS) of Pebble, x be a parametrised circuit description C , and y be some input data. Then p_1 corresponds to the abstraction mechanism, which in this case can be described using a flattening procedure (FP). Similarly p_2 corresponds to the semantics (FS) of the flattened description on the data as shown in Fig. 3.

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:

```

circuit ::= BLOCK main
          [idin1 : typein1, ..., idinn : typeinn]
          [idout1 : typeout1, ..., idoutm : typeoutm]
          dec1; ... ; decj
          BEGIN stmts END
type    ::= WIRE
dec     ::= VAR id : type
stmts   ::= stmt1; ... ; stmtk
stmt    ::= id℘ [id1, ..., idn] [id1, ..., idm]

```

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 \mathbf{a} , so that primitive objects can be of any type; it allows us to deal with both bit-level descriptions and word-level descriptions.

data wire $\mathbf{a} = \text{Undefined} \mid \text{Defined } \mathbf{a}$

To deal with sequential circuits, the datatype \mathbf{a} can be lifted to the stream domain [JoS88]. A delay primitive can then be included to cover sequential designs.

The structural operational semantics rules for Flattened Pebble, defined by \rightarrow transitions [NiN92], are given in Fig. 4. A local environment ρ maps identifiers to their wires. The operator \oplus denotes relational overwriting. 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 be passed to subsequent gate calls as defined in the syntax for Hierarchical Pebble:

```

hcircuit ::= hmain; hblock1; ... ; hblocki
hmain    ::= BLOCK main
          [idin1 : typein1, ..., idinn : typeinn]
          [idout1 : typeout1, ..., idoutm : typeoutm]
          hdec1; ... ; hdecj
          BEGIN hstmts END
hblock   ::= BLOCK id (idgen1, ..., idgenq)
          [idin1 : typein1, ..., idinn : typeinn]
          [idout1 : typeout1, ..., idoutm : typeoutm]
          hdec1; ... ; hdecj
          BEGIN stmts END

```

$$\begin{array}{c}
\frac{\exists j \cdot 1 \leq j \leq n \wedge (\rho \text{ id}_j) = \text{Undefined}}{\delta \vdash \langle \text{id}_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m], \rho \rangle} \\
\rightarrow_{\text{stmt}} \langle \text{id}_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m], \rho \rangle \\
\\
\frac{(\rho \text{ id}_1) = \text{Defined } v_1 \wedge \dots \wedge (\rho \text{ id}_n) = \text{Defined } v_n \quad (\delta \text{ id}_{\mathcal{P}}) (v_1, \dots, v_n) = (v'_1, \dots, v'_m)}{\delta \vdash \langle \text{id}_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m], \rho \rangle} \\
\rightarrow_{\text{stmt}} \rho \oplus \{id'_1 \mapsto \text{Defined } v'_1, \dots, id'_m \mapsto \text{Defined } v'_m\} \\
\\
\frac{\delta \vdash \langle \text{stmt}_i, \rho \rangle \rightarrow_{\text{stmt}} \rho'}{\delta \vdash \langle \text{stmt}_1; \dots \text{stmt}_{i-1}; \text{stmt}_i; \text{stmt}_{i+1}; \dots \text{stmt}_k, \rho \rangle} \\
\rightarrow_{\text{stmts}} \langle \text{stmt}_1; \dots \text{stmt}_{i-1}; \text{stmt}_{i+1}; \dots \text{stmt}_k, \rho' \rangle \\
\\
\frac{\delta \vdash \langle \text{stmt}_i, \rho \rangle \rightarrow_{\text{stmt}} \langle \text{stmt}_i, \rho \rangle}{\delta \vdash \langle \text{stmt}_1; \dots \text{stmt}_{i-1}; \text{stmt}_i; \text{stmt}_{i+1}; \dots \text{stmt}_k, \rho \rangle} \\
\rightarrow_{\text{stmts}} \langle \text{stmt}_1; \dots \text{stmt}_{i-1}; \text{stmt}_i; \text{stmt}_{i+1}; \dots \text{stmt}_k, \rho \rangle \\
\\
\delta \vdash \langle [], \rho \rangle \rightarrow_{\text{stmts}} \rho \\
\\
\begin{array}{l}
\rho_1 = \{id_{in_1} \mapsto \text{Defined } v_1, \dots, id_{in_n} \mapsto \text{Defined } v_n\} \\
\rho_2 = \{id_{out_1} \mapsto \text{Undefined}, \dots, id_{out_m} \mapsto \text{Undefined}\} \\
\rho_3 = \{id_1 \mapsto \text{Undefined}, \dots, id_j \mapsto \text{Undefined}\} \\
\rho = \rho_1 \oplus \rho_2 \oplus \rho_3
\end{array} \\
\delta \vdash \langle \text{stmts}, \rho \rangle \rightarrow_{\text{stmts}} \rho' \\
\\
\delta \vdash \left\langle \left(\begin{array}{l}
\text{BLOCK main} \\
[id_{in_1} : \text{WIRE}, \dots, id_{in_n} : \text{WIRE}] \\
[id_{out_1} : \text{WIRE}, \dots, id_{out_m} : \text{WIRE}] \\
\text{VAR } id_1 : \text{WIRE}; \dots \text{VAR } id_j : \text{WIRE} \\
\text{BEGIN stmts END} \\
\rightarrow_{\text{main}} [(\rho' id_{out_1}), \dots, (\rho' id_{out_m})]
\end{array} \right), [v_1, \dots, v_n] \right\rangle
\end{array}$$

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

```

hdec ::= VAR id : type | VAR id : NUM
hstmts ::= hstmt1; ... ; hstmtk
hstmt ::= idP [id1, ..., idn] [id1, ..., idm]
        | id (exp1, ..., expq) [id1, ..., idn] [id1, ..., idm]
        | GENERATE FOR id = exp1 .. exp2
          BEGIN stmts END
exp ::= id | n | exp1 bop exp2 | uop exp

```

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.

Loops describe the structure of a circuit and are not incremental in operation as in imperative programming languages. Loops require two passes. The first pass creates a list of pairs binding value environments to their statement bodies. Therefore each loop iteration creates a distinct σ that will be used to give meaning to that particular instance of the loop body. The three rules for statement pairs show how progress is made while evaluating loop bodies until the final environment ρ has been calculated. Statement pairs can be evaluated in any order and in parallel.

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

$$\begin{array}{c}
\Gamma, \delta \vdash \langle [], \rho \rangle \Rightarrow_{stpairs} \rho \\
\hline
\frac{\Gamma, \delta, \sigma_i \vdash \langle stmts_i, \rho \rangle \Rightarrow_{stmts} \rho'}{\Gamma, \delta \vdash \langle [(\sigma_1, stmts_1), \dots, (\sigma_{i-1}, stmts_{i-1}), (\sigma_i, stmts_i), (\sigma_{i+1}, stmts_{i+1}), \dots, (\sigma_n, stmts_n)], \rho \rangle \Rightarrow_{stpairs} \langle [(\sigma_1, stmts_1), \dots, (\sigma_{i-1}, stmts_{i-1}), (\sigma_{i+1}, stmts_{i+1}), \dots, (\sigma_n, stmts_n)], \rho' \rangle} \\
\hline
\frac{\Gamma, \delta, \sigma_i \vdash \langle stmts_i, \rho \rangle \Rightarrow_{stmts} \langle stmts'_i, \rho' \rangle}{\Gamma, \delta \vdash \langle [(\sigma_1, stmts_1), \dots, (\sigma_{i-1}, stmts_{i-1}), (\sigma_i, stmts_i), (\sigma_{i+1}, stmts_{i+1}), \dots, (\sigma_n, stmts_n)], \rho \rangle \Rightarrow_{stpairs} \langle [(\sigma_1, stmts_1), \dots, (\sigma_{i-1}, stmts_{i-1}), (\sigma_i, stmts'_i), (\sigma_{i+1}, stmts_{i+1}), \dots, (\sigma_n, stmts_n)], \rho' \rangle} \\
\hline
\frac{\mathcal{E}_\sigma \llbracket exp_1 \rrbracket > \mathcal{E}_\sigma \llbracket exp_2 \rrbracket}{\langle stmts, \sigma, exp_1, exp_2 \rangle \Rightarrow_{pairs} []} \\
\hline
\frac{\mathcal{E}_\sigma \llbracket exp_1 \rrbracket \leq \mathcal{E}_\sigma \llbracket exp_2 \rrbracket \quad \langle stmts, \sigma, exp_1 + 1, exp_2 \rangle \Rightarrow_{pairs} ps}{\langle stmts, \sigma, exp_1, exp_2 \rangle \Rightarrow_{pairs} [(\sigma \oplus \{id_{index} \mapsto \mathcal{E}_\sigma \llbracket exp_1 \rrbracket\}, stmts)] \# ps} \\
\hline
\frac{\langle stmts, \sigma, exp_1, exp_2 \rangle \Rightarrow_{pairs} ps \quad \Gamma, \delta \vdash \langle ps, \rho \rangle \Rightarrow_{stpairs} \rho'}{\Gamma, \delta, \sigma \vdash \left\langle \left(\begin{array}{l} \text{GENERATE FOR } id_{index} = exp_1 \dots exp_2 \\ \text{BEGIN } stmts \text{ END} \end{array} \right), \rho \right\rangle \Rightarrow_{stmt} \rho} \\
\hline
(\Gamma \text{ id}) = \left(\begin{array}{l} \vdots \\ \text{BLOCK } id \text{ } (id_{gen_1}, \dots, id_{gen_q}) \quad [id_{in_1} : \text{WIRE}, \dots, id_{in_n} : \text{WIRE}] \\ \quad [id_{out_1} : \text{WIRE}, \dots, id_{out_m} : \text{WIRE}] \\ \text{VAR } id_{index_1} : \text{NUM}; \dots \text{ VAR } id_{index_j} : \text{NUM}; \\ \text{VAR } id_{local_1} : \text{WIRE}; \dots \text{ VAR } id_{local_k} : \text{WIRE} \\ \text{BEGIN } stmts \text{ END} \\ \sigma_1 = \{id_{gen_1} \mapsto \mathcal{E}_\sigma \llbracket e_1 \rrbracket, \dots, id_{gen_q} \mapsto \mathcal{E}_\sigma \llbracket e_q \rrbracket\} \\ \rho_1 = \{id_{in_1} \mapsto (\rho \text{ id}_1), \dots, id_{in_n} \mapsto (\rho \text{ id}_n)\} \\ \rho_2 = \{id_{out_1} \mapsto \text{Undefined}, \dots, id_{out_m} \mapsto \text{Undefined}\} \\ \rho_3 = \{id_{local_1} \mapsto \text{Undefined}, \dots, id_{local_k} \mapsto \text{Undefined}\} \\ \rho' = \rho_1 \oplus \rho_2 \oplus \rho_3 \end{array} \right) \\
\hline
\frac{\Gamma, \delta, \sigma_1 \vdash \langle stmts, \rho' \rangle \Rightarrow_{stmts} \rho''}{\Gamma, \delta, \sigma \vdash \langle id \text{ } (e_1, \dots, e_q) \text{ } [id_1, \dots, id_n] \text{ } [id'_1, \dots, id'_m], \rho \rangle \Rightarrow_{stmt} \rho \oplus \{id'_1 \mapsto (\rho'' \text{ id}_{out_1}), \dots, id'_m \mapsto (\rho'' \text{ id}_{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. The $\#$ operator denotes list concatenation

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.

3.3. Flattening procedure

A Hierarchical Pebble description can be flattened to produce Flattened Pebble 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

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

Fig. 6. Semantics of the main block in Pebble

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:

$$\forall id \cdot \rho id = \rho_d (\mu id) \quad (1)$$

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 or variant 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 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 primitive gate calls forms the body of the flattened main block and the derived intermediate wires are declared local to this 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;
END;

```

$$\begin{array}{c}
\Gamma, \mu, \sigma \vdash \langle id_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m] \rangle \\
\Downarrow_{stmt} ([id_{\mathcal{P}} [\mu id_1, \dots, \mu id_n] [\mu id'_1, \dots, \mu id'_m]], []) \\
\\
(\alpha (\Gamma id)) = \left(\begin{array}{l}
\text{BLOCK } id (id_{gen_1}, \dots, id_{gen_q}) [id_{in_1}:WIRE, \dots, id_{in_n}:WIRE] \\
\quad [id_{out_1}:WIRE, \dots, id_{out_m}:WIRE] \\
\text{VAR } id_{index_1} : \text{NUM}; \dots \text{VAR } id_{index_j} : \text{NUM} \\
\text{VAR } id_{local_1} : \text{WIRE}; \dots \text{VAR } id_{local_k} : \text{WIRE} \\
\text{BEGIN } stmts \text{ END} \\
\mu_1 = \{id_{in_1} \mapsto (\mu id_1), \dots, id_{in_n} \mapsto (\mu id_n)\} \\
\mu_2 = \{id_{out_1} \mapsto (\mu id'_1), \dots, id_{out_m} \mapsto (\mu id'_m)\} \\
\mu_3 = \{id_{local_1} \mapsto id_{local_1}, \dots, id_{local_k} \mapsto id_{local_k}\} \\
\sigma_1 = \{id_{gen_1} \mapsto \mathcal{E}_{\sigma}[e_1], \dots, 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} \right) \\
\hline
\Gamma, \mu, \sigma \vdash \langle id (e_1, \dots, e_q) [id_1, \dots, id_n] [id'_1, \dots, id'_m] \rangle \\
\Downarrow_{stmt} (stmts', [\text{VAR } id_{local_1} : \text{WIRE}; \dots; \text{VAR } id_{local_k} : \text{WIRE}] \# locals') \\
\\
\frac{\mathcal{E}_{\sigma}[exp_1] > \mathcal{E}_{\sigma}[exp_2]}{\Gamma, \mu, \sigma \vdash \langle (\text{GENERATE FOR } id_{index}=exp_1 \dots exp_2 \text{ BEGIN } stmts \text{ END}) \rangle \Downarrow_{stmt} ([], [])} \\
\\
\frac{\mathcal{E}_{\sigma}[exp_1] \leq \mathcal{E}_{\sigma}[exp_2] \quad \Gamma, \mu, \sigma \oplus \{id_{index} \mapsto \mathcal{E}_{\sigma}[exp_1]\} \vdash \langle stmts \rangle \Downarrow_{stmts} (stmts', locals')}{\Gamma, \mu, \sigma \vdash \langle \langle (\text{GENERATE FOR } id_{index}=(exp_1+1) \dots exp_2) \rangle \rangle \Downarrow_{stmt} (stmts'', locals'')} \\
\hline
\Gamma, \mu, \sigma \vdash \langle \langle (\text{GENERATE FOR } id_{index}=exp_1 \dots exp_2) \rangle \rangle \\
\Downarrow_{stmt} (stmts' \# stmts'', locals' \# locals'') \\
\\
\frac{\Gamma, \mu, \sigma \vdash \langle stmt_1 \rangle \Downarrow_{stmt} (stmts_1, locals_1) \dots \Gamma, \mu, \sigma \vdash \langle stmt_n \rangle \Downarrow_{stmt} (stmts_n, locals_n)}{\Gamma, \mu, \sigma \vdash \langle stmt_1; \dots; stmt_n \rangle \Downarrow_{stmts} (stmts_1 \# \dots \# stmts_n, locals_1 \# \dots \# locals_n)}
\end{array}$$

Fig. 7. Transition rules for flattening Pebble statements, based on \Downarrow rules for $stmts$ and $stmt$. The $\#$ operator concatenates together statement lists and local declaration lists

```

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. Fig. 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.

4. Verifying flattening procedure

We can now present the main correctness theorem for flattening hierarchical blocks and `GENERATE-FOR` loops. This result, given by Eq. 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 Eq. 1. At each syntactic level, they show how

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

Fig. 8. Transition rules for flattening the main block

$$\begin{array}{c}
\vdots \\
\hline
\Gamma, \mu', \sigma_1 \vdash \left\langle \begin{array}{l} \text{connect [vin1] [inter1(0)];} \\ \text{GENERATE FOR } \dots \\ \vdots \\ \text{connect [inter1(n1)] [vout1]} \end{array} \right\rangle \Downarrow_{stmts} ([\text{connect [x] [inter1(0)];} \\ \text{not [inter1(0)] [inter1(1)];} \\ \text{not [inter1(1)] [inter1(2)];} \\ \text{connect [inter1(2)] [y],} \\ []) \\
\sigma_1 = \{n1 \mapsto 2\} \\
\mu_1 = \{\text{vin1} \mapsto x\} \quad \mu_2 = \{\text{vout1} \mapsto y\} \quad \mu_3 = \{i1 \mapsto i1\} \\
\mu' = \mu_1 \oplus \mu_2 \oplus \mu_3 \\
(\alpha(\Gamma \text{ notrow})) = \text{BLOCK notrow (n1) [vin1:WIRE] [vot1:WIRE]} \\
\quad \text{VAR inter1:VECTOR (n1..0) OF WIRE;} \\
\quad \text{VAR i1:NUM} \\
\quad \text{BEGIN} \\
\quad \quad \text{connect [vin1] [inter1(0)];} \\
\quad \quad \text{GENERATE FOR i1=0..n1-1} \\
\quad \quad \quad \text{BEGIN not [inter1(i1)] [inter1(i1+1)] END;} \\
\quad \quad \quad \text{connect [inter1(n1)] [vot1]} \\
\quad \quad \text{END;} \\
\hline
\Gamma, \{x \mapsto x, y \mapsto y\}, \{\} \vdash (\text{notrow (2) [x] [y]}) \\
\quad \Downarrow_{stmt} ([\text{connect [x] [inter1(0)];} \\
\quad \quad \text{not [inter1(0)] [inter1(1)];} \\
\quad \quad \text{not [inter1(1)] [inter1(2)];} \\
\quad \quad \text{connect [inter1(2)] [y], []} \\
\quad \quad [\text{VAR inter1:VECTOR (2..0) OF WIRE}])
\end{array}$$

Fig. 9. Fragment of the proof tree for flattening notrow, where α is used to rename variables and μ maps local variables to those of the block call

the sequence of outputs generated by the hierarchical definition can be calculated by first flattening the term and then using the simplified semantics.¹

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 $[v_1, \dots, v_n]$, 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:

$$\begin{aligned} \Gamma, \delta \vdash \langle \text{main}, [v_1, \dots, v_n] \rangle &\Rightarrow_{\text{main}} [v'_1, \dots, v'_m] \\ \implies \Gamma \vdash \text{main} \Downarrow_{\text{main}} \text{main}' \\ \wedge \delta \vdash \langle \text{main}', [v_1, \dots, v_n] \rangle &\rightarrow_{\text{main}} [v'_1, \dots, v'_m] \end{aligned} \quad (2)$$

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 also created and bound within μ . 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 Eq. 1, to hold for wire environments:

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

This lemma can be proved by induction on the length of derivation sequences using Lemma 4; 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 :

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

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:

¹ With the addition of conditional statements, recursive block definitions can result in non-terminating programs. In these cases the flattening procedure will also fail to terminate

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

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 [Den00, PfS99] or translated into Horn clauses via a metalanguage such as Typol [Des84]. 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 [ACG92] 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: 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 process. The original non-verified implementation, in contrast, contains a single procedure which performs both variable renaming and flattening. The mingling of the two functions, however, leads to situations where the renaming process is deeply nested within the unfolding procedure, leaving little scope for further optimisations. In comparison, the new version is amenable to further optimisations, such as the use of de Bruijn indices to avoid the costly rename function [Han94]. 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. One explanation is that the verification process often reveals insights about a design, and such insights can often be used to improve its efficiency.

6. Placement information

This section describes both explicit and relative placement information for Pebble. We demonstrate both placement styles on a simple pattern matcher. Precise control of library layout using placement information is especially rewarding in two situations. First, such information is particularly effective for regular circuits, where conventional placement algorithms may not be able to fully exploit the circuit structure. Second, controlling placement is desirable for reconfigurable circuits to minimize reconfiguration time, since components at identical locations common to two successive configurations do not need to be reconfigured. Such optimisation has been included in recent design tools for reconfigurable applications [SLC00].

Pebble adopts the convention “AT (x, y) ” to denote the placement of a block at a location with coordinates (x, y) as shown in Fig. 11. Flattening proceeds as before, only the rule for primitive gate calls of Figure 7 needs to be modified:

$$\begin{aligned}
& \Gamma, \mu, \sigma \vdash \langle id_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m] \text{ AT } (e_1, e_2) \rangle \\
& \Downarrow_{stms} ([id_{\mathcal{P}} [\mu \ id_1, \dots, \mu \ id_n] [\mu \ id'_1, \dots, \mu \ id'_m] \text{ AT } (\mathcal{E}_{\sigma} [e_1], \mathcal{E}_{\sigma} [e_2]), [])
\end{aligned}$$

```

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,genms,parms1,parms2,decs,stmts)) = rename (fetch gamma bnm)
    sigma1 = zip genms (map (\ e -> eval_exp e sigma) gens)
    mu1 = zip parms1 (map (fetch mu) args1)
    mu2 = zip parms2 (map (fetch 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

```

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)] AT (i,0)
  END
  END;

```

Fig. 11. A description of an array of multiplexors (Fig. 1) in Pebble with explicit placement coordinates

```

BBlock ::= BLOCK id (idgen1, ..., idgenq)
           [idin1:typein1, ..., idinn:typeinn]
           [idout1:typeout1:typeoutm, ..., idoutm]
           dec1; ...; decj
           BEGIN bes END

bes ::= idP [id1, ..., idn] [id1, ..., idm]
        | id (exp1, ..., expq) [id1, ..., idn] [id1, ..., idm]
        | BESIDE (cstmt1; ...; cstmtn)
        | BELOW (cstmt1; ...; cstmtn)
        | BESIDE FOR id = exp1..exp2 BEGIN bes END
        | BELOW FOR id = exp1..exp2 BEGIN bes END

cstmt ::= GENERATE IF exp THEN bes
        | bes

```

Fig. 12. Syntax of Beside and Below Pebble with conditionals

While such placement information helps to optimize the layout, it is usually tedious and error-prone to specify. We have therefore developed high-level descriptions for placement constraints, abstracting away the low-level details. These descriptions are compile-time directives for the Pebble compiler to project coordinates onto designs, generating a tree representing placement possibilities.

The two main descriptions, shown in Fig. 12, are BESIDE, which places two or more blocks beside each other, and BELOW, which places blocks vertically. These descriptions allow blocks to be placed *relatively* to each other, without the user providing the coordinates of their locations.

For a case study, consider a simple pattern matcher design that matches a string of bits in a serial data stream. The design features a regular array of 1-bit pattern matching cells. The performance and area utilisation can be optimised by controlling the layout using placement information. We show that the BESIDE and BELOW operators allow a compact and efficient layout to be captured in a simple way.

Figure 13a shows a pattern matcher design that matches strings with a bit width of 1. The pattern is stored in the lower shift register when *load*=1. This register contains d-type flip-flops with an enable (labelled *fd_e*). The data moves through the d-type flip-flops of the top shift register (labelled *fd*). The data and the pattern are compared by the combinational logic implemented by the 3-input lookup tables (labelled *lut3*). The output, *match*, is asserted when a match is found.

To layout this design, we assume that the primitives *fd*, *fd_e* and *lut3* occupy a single location. The layout corresponds to the positions of the primitives shown in Fig. 13a. This can be captured with BESIDE and BELOW by composing a 1-bit matcher using BELOW and then by laying it out as a row using BESIDE. A multi-bit pattern matcher can then be composed as a column using BELOW to stack up the 1-bit rows. The match results must be combined using a further AND gate to obtain the final result.

If the match pattern is constant, then a specialised implementation can be produced by removing the pattern shift register and simplifying the combinational logic by boolean optimisation. The combinational logic can then be mapped to a 2-input lookup table as shown in Fig. 13b. The match pattern is encoded into the lookup tables using an AND gate for a 1, and an AND gate with an inverted input for a 0. When implemented on an FPGA, the match pattern can be changed by reconfiguring the data in the lookup tables. This optimisation can be used to reduce the amount of resources required to implement the design. For a multi-bit pattern matcher, the layout can be compacted after optimisation by closing the gaps created by the removal of the pattern shift register.

Figure 14 shows a parametrised Pebble description that can be used to implement either the full pattern matcher (Fig. 13a or the specialised pattern matcher Fig. 13b): parameter *w* is the bit-width of the data; *n* is the length of the match pattern; *specialise* selects between the full and specialised implementations; and *pattern* is the match pattern for the specialised implementation. When the BESIDE-BELOW description is converted to a coordinate description, the coordinates are assigned within the conditionals such that a compact layout is achieved for each of the alternate implementations. This is achieved using partial evaluation techniques [JGS93] and is discussed further in Sect. 9.

Figure 15 shows the corresponding Pebble description using coordinates. In this description, the position of each instance is provided by a set of coordinates comprising of symbolic arithmetic expressions given in terms of the design parameters. The mechanism to compact the layout when evaluating the conditionals must be provided explicitly. Using the BESIDE and BELOW operators, this compaction is provided for free, hence removing the need to provide an otherwise tedious and error-prone layout description. The next section describes the process for

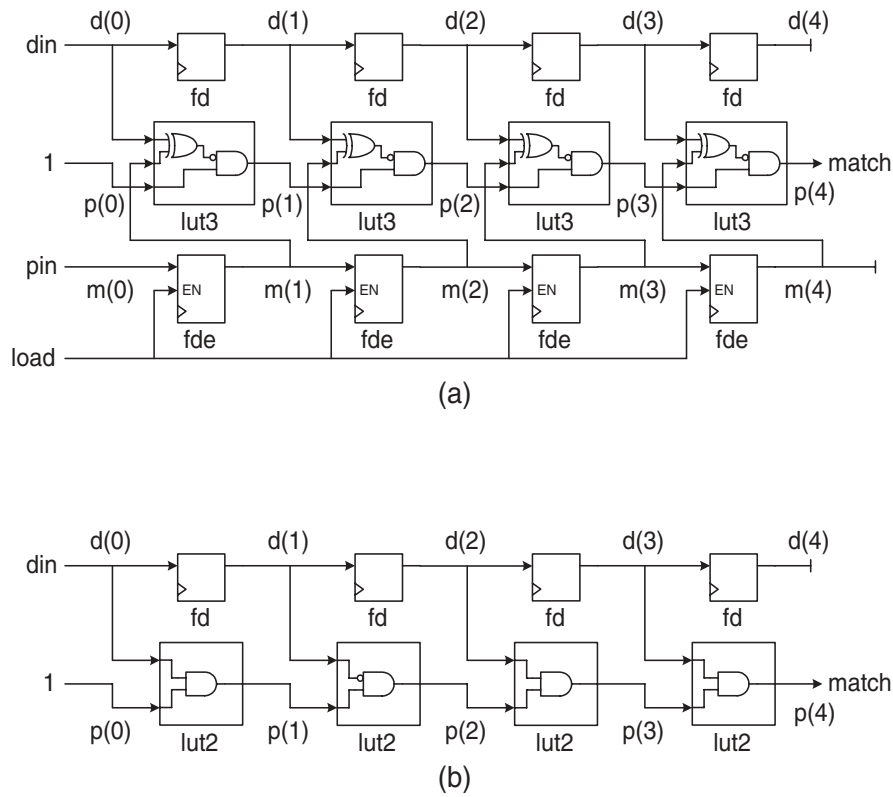


Fig. 13. Two 1-bit pattern matcher implementations with $n = 4$. (a) Non-specialised design ($\text{specialise}=0$). (b) Specialised design ($\text{specialise}=1$) with $\text{pattern}=[1, 1, 0, 1]$

mapping a description with placement given by BESIDE and BELOW to a description with placement given by coordinates.

7. Compiling placement descriptions

In order to project a coordinate scheme onto a Beside–Below Pebble statement, we use an environment Γ_B mapping block names to their syntactical definitions, an environment ϕ mapping block names to their sizes, and a placement function \mathcal{P} (Fig. 16). Block sizes are functions that take the symbolic arguments of a block and return its symbolic width and height. The placement function \mathcal{P} is used to position blocks within their immediate context; it maps an abstract coordinate scheme onto a statement. It returns a tuple of three components: a sequence of statements unfolded by the rules of BESIDE and BELOW, the dimensions of the block associated with the statement, and an updated block size environment ϕ .

The placement of blocks is achieved locally to derive suitable symbolic locations. Symbolic addresses are calculated using the given (x, y) expressions and the function ‘ f ’. For placing single blocks, an identity function is used for ‘ f ’. For designs that derive repeated positions in BESIDE FOR and BELOW FOR loops, we create two new local placement functions, $g1$ and $g2$, that ‘ f ’ could become. These two functions do not depend on the nesting level of the statement, but only on the given start position of the loop. Our model does not include space for wiring: it is assumed that wiring resources are orthogonal to the network of logic blocks and have no effects on them, or that the effects of routing between logic blocks are captured within the blocks themselves.

A coordinate scheme is projected onto a Beside–Below statement in the following manner. A primitive block of width $\text{across}(id_{\mathcal{P}})$ and height $\text{upwards}(id_{\mathcal{P}})$ is positioned according to its placement function and dimension.

```

BLOCK pmatch (w, n, specialise, pattern)
  [din, pin : VECTOR(w-1..0) OF WIRE;
   load, clk : WIRE]
  [match : VECTOR(w-1..0) OF WIRE]
  VAR i, j;
  VAR d : VECTOR(n..0) OF VECTOR(w-1..0) OF WIRE;
  VAR p : VECTOR(n..0) OF VECTOR(w-1..0) OF WIRE
  VAR m : VECTOR(n..0) OF VECTOR(w-1..0) OF WIRE
BEGIN
  BELOW FOR j=0..w-1
  BEGIN
    connect [d(0)(j)] [din(j)];
    connect [m(0)(j)] [pin(j)];
    connect [match(j)] [p(size)(j)];

    BESIDE (
      constant (1) [] [p(0)(j)];

      BESIDE FOR i=0..n-1
      BEGIN
        BELOW (
          GENERATE IF specialise = 0
          THEN fde [m(i)(j), clk, load] [m(i+1)(j)];
          lut3 (132) [d(i)(j), p(i)(j), m(i+1)(j)]
            [p(i+1)(j)]
          END;
          GENERATE IF specialise = 1
          THEN
            GENERATE IF pattern(i) = 0 THEN
              lut2 (4) [d(i)(j), p(i)(j)] [p(i+1)(j)]
            ELSE
              lut2 (8) [d(i)(j), p(i)(j)] [p(i+1)(j)]
            END
          END
          fd [d(i)(j), clk] [d(i+1)(j)] )
        END )
      END
    )
  END;
END;

```

Fig. 14. Pebble description of the pattern matcher design with placement given by the BESIDE and BELOW operators. Parameter n is the length of the pattern. *constant (1)* denotes a constant generator producing 1. *lut2* and *lut3* are lookup tables, and *fd* and *fde* are d-type flip flops. A specialised implementation is generated when *specialise=1*, otherwise a full implementation is generated. When specialised, the use of the *BELOW* ensures that the design is compacted. Parameter *pattern* is the match pattern when the design is specialised. The AND gates required to produce the final result are not included in this description

The size expression of composite blocks is calculated by applying the generic expressions to the block's size stored in ϕ . If the size expression is unknown, then it is derived using \mathcal{PB} . Coordinates are projected onto a row of beside terms by adding previous widths together. The final size of the BESIDE statement is the sum of each width and the maximum height of all subterms. Similarly for the BELOW statement.

For loops, the position of each loop body depends on the iteration index and the size of the body. Initially, we do not know the size of the loop body so we create a new identifier using the function \mathcal{NV} , and replace it with the value once it is known. The concealed function \mathcal{NV} creates a distinct new identifier each time it is called. This method works because the place holder variables will not be required until after the size of the block is known. The position of each repeated subterm is calculated using a new placement function.

The size of a Beside–Below block is calculated from the size of its statement body using \mathcal{P} and the default identity placement function f . The resulting dimensions (acc, up) are parametrised by the block's generic variables (gid_1, \dots, gid_j), as shown in the lambda expression of Fig. 17. This expression denotes the size of the block when applied to a list of values; it is bound to the block's name and added to the updated size environment ϕ' .

We can use the above definitions to prove the correctness of various source to source transformations. As an example, consider the composition of two BESIDE statements:

$$\begin{aligned}
& \mathcal{P}_{\phi} \Gamma_B \llbracket \text{BESIDE}(a; \text{BESIDE}(b; c)) \rrbracket (x, y) f \\
& = \mathcal{P}_{\phi} \Gamma_B \llbracket \text{BESIDE}(a; b; c) \rrbracket (x, y) f
\end{aligned}$$

```

BLOCK pmatch (x, y, w, n, specialise, pattern)
  [din, pin : VECTOR(w-1..0) OF WIRE;
   load, clk : WIRE]
  [match : VECTOR(w-1..0) OF WIRE]

  VAR i, j;
  VAR d : VECTOR(n..0) OF VECTOR(w-1..0) OF WIRE;
  VAR p : VECTOR(n..0) OF VECTOR(w-1..0) OF WIRE
  VAR m : VECTOR(n..0) OF VECTOR(w-1..0) OF WIRE
BEGIN
  GENERATE FOR j=0..w-1
  BEGIN
    connect [d(0)(j)] [din(j)];
    connect [m(0)(j)] [pin(j)];
    connect [match(j)] [p(size)(j)];

    GENERATE IF specialise = 0 THEN
      constant (1) [] [p(0)(j)] AT (x,y+(j*3));
    END;
    GENERATE IF specialise = 1 THEN
      constant (1) [] [p(0)(j)] AT (x,y+(j*2));
    END;

    GENERATE FOR i=0..n-1
    BEGIN
      GENERATE IF specialise = 0
      THEN
        fde [m(i)(j), clk, load]
          [m(i+1)(j)] AT (x+i+1,y+(j*3));
        lut3 (132) [d(i)(j), p(i)(j), m(i+1)(j)]
          [p(i+1)(j)] AT (x+i+1,y+(j*3)+1);
        fd [d(i)(j), clk]
          [d(i+1)(j)] AT (x+i+1,y+(j*3)+2)
      END;
      GENERATE IF specialise = 1
      THEN
        GENERATE IF pattern(i) = 0 THEN
          lut2 (4) [d(i)(j), p(i)(j)]
            [p(i+1)(j)] AT (x+i+1,y+(j*2))
        ELSE
          lut2 (8) [d(i)(j), p(i)(j)]
            [p(i+1)(j)] AT (x+i+1,y+(j*2))
        END;
        fd [d(i)(j), clk]
          [d(i+1)(j)] AT (x+i+1,y+(j*2)+1)
      END
    END
  END
END
END;

```

Fig. 15. Pebble description of the pattern matcher design with placement given by explicit placement expressions in the form of symbolic placement constraints. Parameters x and y are the coordinates of the origin of the *pmatch* block

A proof can be obtained by unfolding the LHS twice using \mathcal{P} , rearranging the resulting expression, and then folding \mathcal{P} to arrive at the RHS.

8. Verifying placement compilation

The verification of the compilation scheme for designs with relative placement, specified by the mapping \mathcal{P} in the preceding section (Fig. 16), involves three steps. The first step is to specify a flattened version of Pebble with explicit placement coordinates. This has been achieved with the \rightarrow rules of Fig. 4. The second step is to specify the semantics of descriptions with relative placement using a function \mathcal{B} , and to specify the semantics of descriptions with explicit placement using \Downarrow_{stms} . The third step is to show that the composition of \mathcal{P} and \Downarrow_{stms} corresponds to \mathcal{B} , as illustrated by the commuting diagram in Fig. 3 with \mathcal{FP} and \mathcal{PS} becoming, respectively \mathcal{P} and \mathcal{B} .

To assign meaning to descriptions with relative placement, we use the function \mathcal{B} for mapping such descriptions with relative placement directly to flattened descriptions in which primitive blocks have numerical placement

$$\begin{aligned}
& \mathcal{P} :: \text{SizeEnv} \rightarrow \text{BBlockEnv} \rightarrow \text{bes} \rightarrow (\text{exp} \times \text{exp}) \rightarrow \text{FuncPos} \rightarrow (\text{hstmts} \times (\text{exp} \times \text{exp}) \times \text{SizeEnv}) \\
& \mathcal{P}_{\phi \Gamma_B} \llbracket \text{id}_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m] \rrbracket (x, y) f \\
& \quad = \text{let } (xpos, ypos) = f(x, y) \\
& \quad \quad \text{in } ([id_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m] \text{ AT } (xpos, ypos)], (\text{across}(id_{\mathcal{P}}), \text{upwards}(id_{\mathcal{P}})), \phi) \\
& \mathcal{P}_{\phi \Gamma_B} \llbracket \text{id } (exp_1, \dots, exp_j) [id_1, \dots, id_n] [id'_1, \dots, id'_m] \rrbracket (x, y) f \\
& \quad = \text{if } (id \in (\text{dom } \phi)) \\
& \quad \quad \text{then} \\
& \quad \quad \quad \text{let } (acc, up) = (\phi \text{ id } (exp_1, \dots, exp_j)) \\
& \quad \quad \quad \quad (xpos, ypos) = f(x, y) \\
& \quad \quad \quad \text{in } ([id(xpos, ypos, exp_1, \dots, exp_j) [id_1, \dots, id_n] [id'_1, \dots, id'_m]], \\
& \quad \quad \quad \quad (acc, up), \phi) \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{let } \phi' = \mathcal{P}\mathcal{B}_{\phi \Gamma_B}(\Gamma_B \text{ id}) \\
& \quad \quad \quad \quad (acc, up) = (\phi' \text{ id } (exp_1, \dots, exp_j)) \\
& \quad \quad \quad \quad (xpos, ypos) = f(x, y) \\
& \quad \quad \quad \text{in } ([id(xpos, ypos, exp_1, \dots, exp_j) [id_1, \dots, id_n] [id'_1, \dots, id'_m]], \\
& \quad \quad \quad \quad (acc, up), \phi') \\
& \mathcal{P}_{\phi \Gamma_B} \llbracket \text{BESIDE}(bes_1; \dots; bes_n) \rrbracket (x, y) f \\
& \quad = \text{let } (stmts_1, (acc_1, up_1), \phi_1) = \mathcal{P}_{\phi \Gamma_B} \llbracket bes_1 \rrbracket (x, y) f \\
& \quad \quad (stmts_2, (acc_2, up_2), \phi_2) = \mathcal{P}_{\phi_1 \Gamma_B} \llbracket bes_2 \rrbracket (x + acc_1, y) f \\
& \quad \quad \quad \vdots \\
& \quad \quad (stmts_n, (acc_n, up_n), \phi_n) = \mathcal{P}_{\phi_{n-1} \Gamma_B} \llbracket bes_n \rrbracket (x + acc_1 + \dots + acc_{n-1}, y) f \\
& \quad \quad \text{in } (stmts_1 \# \dots \# stmts_n, (acc_1 + \dots + acc_n, \max(up_1, \dots, up_n)), \phi_n) \\
& \mathcal{P}_{\phi \Gamma_B} \llbracket \text{BELOW}(bes_1; \dots; bes_n) \rrbracket (x, y) f \\
& \quad = \text{let } (stmts_1, (acc_1, up_1), \phi_1) = \mathcal{P}_{\phi \Gamma_B} \llbracket bes_1 \rrbracket (x, y) f \\
& \quad \quad (stmts_2, (acc_2, up_2), \phi_2) = \mathcal{P}_{\phi_1 \Gamma_B} \llbracket bes_2 \rrbracket (x, y + up_1) f \\
& \quad \quad \quad \vdots \\
& \quad \quad (stmts_n, (acc_n, up_n), \phi_n) = \mathcal{P}_{\phi_{n-1} \Gamma_B} \llbracket bes_n \rrbracket (x, y + up_1 + \dots + up_{n-1}) f \\
& \quad \quad \text{in } (stmts_1 \# \dots \# stmts_n, (\max(acc_1, \dots, acc_n), up_1 + \dots + up_n), \phi_n) \\
& \mathcal{P}_{\phi \Gamma_B} \llbracket \text{BESIDE FOR } id = exp_1 \dots exp_2 \text{ BEGIN } bes \text{ END} \rrbracket (x, y) f \\
& \quad = \text{let } xoffset = \mathcal{NV}() \\
& \quad \quad g1(x, y) = (x + (id - exp_1) \times xoffset, y) \\
& \quad \quad (stmts, (acc, up), \phi') = \mathcal{P}_{\phi \Gamma_B} \llbracket bes \rrbracket (x, y) g1 \\
& \quad \quad stmts' = (\lambda xoffset \cdot stmts) acc \\
& \quad \quad \text{in } ([FOR id = exp_1 \dots exp_2 \text{ BEGIN } stmts' \text{ END}], \\
& \quad \quad \quad (acc \times (exp_2 - exp_1 + 1), up), \phi') \\
& \mathcal{P}_{\phi \Gamma_B} \llbracket \text{BELOW FOR } id = exp_1 \dots exp_2 \text{ BEGIN } bes \text{ END} \rrbracket (x, y) f \\
& \quad = \text{let } yoffset = \mathcal{NV}() \\
& \quad \quad g2(x, y) = (x, y + (id - exp_1) \times yoffset) \\
& \quad \quad (stmts, (acc, up), \phi') = \mathcal{P}_{\phi \Gamma_B} \llbracket bes \rrbracket (x, y) g2 \\
& \quad \quad stmts' = (\lambda yoffset \cdot stmts) up \\
& \quad \quad \text{in } ([FOR id = exp_1 \dots exp_2 \text{ BEGIN } stmts' \text{ END}], \\
& \quad \quad \quad (acc, up \times (exp_2 - exp_1 + 1)), \phi')
\end{aligned}$$

Fig. 16. Compiling descriptions with relative placement to descriptions with explicit placement coordinates constructed symbolically. The # operator concatenates together statement lists

positions (Fig. 18). The \mathcal{B} function begins at the bottom left hand corner of a design, and uses a recursive descent algorithm to position each primitive block.

We can now proceed to the third step: to verify the correctness of \mathcal{P} by structural induction on the syntax of Pebble statements containing relative placement information (Fig. 12). The purpose is to show that, for all cases, the two-stage evaluation process with \mathcal{P} and \downarrow_{stmts} yields the same results as the one where the initial coordinates are provided and the language unfolded in one stage using the function \mathcal{B} . The theorem that corresponds to the commuting diagram in Fig. 3 is as follows:

$$\begin{aligned}
\mathcal{PB} &:: \text{SizeEnv} \rightarrow \text{BBlockEnv} \rightarrow \text{BBlock} \rightarrow \text{SizeEnv} \\
\mathcal{P}_{\phi} \Gamma_B \llbracket \text{BLOCK } id (gid_1, \dots, gid_j) \\
&\quad [id_1:\text{WIRE}, \dots, id_n:\text{WIRE}] [id'_1:\text{WIRE}, \dots, id'_m:\text{WIRE}] \\
&\quad \text{VAR } lid_1, \dots, lid_q; \\
&\quad \text{VAR } id''_1:\text{WIRE}, \dots, id''_p:\text{WIRE}; \\
&\text{BEGIN} \\
&\quad bes \\
&\text{END} \rrbracket = \text{let } f(x, y) = (x, y) \\
&\quad (stmts, (acc, up), \phi') = \mathcal{P}_{\phi} \Gamma_B \llbracket bes \rrbracket (x, y) f \\
&\quad \text{in } \phi' \oplus \{ id \mapsto \lambda(gid_1, \dots, gid_j) \cdot (acc, up) \}
\end{aligned}$$

Fig. 17. An algorithm for calculating the size of a block. The identifiers lid_i and wires id'_j are local to this block

$$\begin{aligned}
\mathcal{P}_{\phi} \Gamma_B \llbracket bes \rrbracket (x, y) f &= (hstmts, (exp_1, exp_2), \phi') \\
\Gamma_C, \mu, \sigma \vdash \langle hstmts \rangle &\Downarrow_{stmts} (stmts, decs) \\
\mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket bes \rrbracket (xn, yn) &= (stmts, decs, (n'_1, n'_2)) \\
\text{where } xn = \mathcal{E}_{\sigma'} \llbracket x \rrbracket \text{ and } yn = \mathcal{E}_{\sigma'} \llbracket y \rrbracket & \\
n'_1 = \mathcal{E}_{\sigma'} \llbracket exp_1 \rrbracket \text{ and } n'_2 = \mathcal{E}_{\sigma'} \llbracket exp_2 \rrbracket &
\end{aligned}$$

Verifying the cases for primitives, block calls, beside and below sequences is straightforward since ‘ f ’ would just be the identity function. For the cases involving loops, we need to consider two possible placement functions, g_1 and g_2 , that are used to substitute f . As the placement of blocks is achieved locally, symbolic addresses are calculated using the given (x, y) expressions and the function f . They provide all that is required to derive suitable symbolic locations. Therefore we only need to consider one level of nesting at a time.

One important point is that explicit placement expressions cannot be fully evaluated until the second stage, \Downarrow_{stmts} , since the values of some of the variables are undefined at design time, when \mathcal{P} is involved. Given that these values are known in the run-time environment σ' , where $\sigma' \subseteq \sigma$, we can relate their symbolic representation to their concrete ones as follows:

$$\begin{aligned}
xn &= \mathcal{E}_{\sigma'} \llbracket x \rrbracket \\
yn &= \mathcal{E}_{\sigma'} \llbracket y \rrbracket
\end{aligned}$$

Since the size of a description involving relative placement is the same for both \mathcal{B} and the composition of \mathcal{P} and \Downarrow_{stmts} , we have

$$\begin{aligned}
n'_1 &= \mathcal{E}_{\sigma'} \llbracket exp_1 \rrbracket \\
n'_2 &= \mathcal{E}_{\sigma'} \llbracket exp_2 \rrbracket
\end{aligned}$$

The above proof outline covers a subset of Pebble that does not include vectors and conditional statements. Our approach can be extended to cover such Pebble constructs, such as the syntax in Fig. 12.

9. Dealing with conditionals and compaction by partial evaluation

This section explains how our approach can be extended to deal with conditionals, and how this extension can be used to support design compaction using partial evaluation.

The use of our guarded command, GENERATE IF, causes problems from a placement perspective as we have to consider both what happens when the guard succeeds and fails. Primitive block calls that occur after a conditional call will be placed differently depending on whether the conditional is true or not. Consider the following example:

```

BESIDE ( a;
        GENERATE IF x=2 THEN b;
        c)

```

In effect it describes two situations. If x is 2 then we can rewrite the above as $\text{BESIDE}(a;b;c)$, otherwise it becomes $\text{BESIDE}(a;c)$. Applying \mathcal{P} to each case will result in differing layouts. Our solution is to ensure that all

$$\begin{aligned}
\mathcal{B} &:: (\mathit{BBlockEnv} \times \mathit{WireEnv} \times \mathit{NumEnv}) \rightarrow \mathit{bes} \rightarrow (\mathit{Num} \times \mathit{Num}) \rightarrow (\mathit{stmts} \times \mathit{Decs} \times (\mathit{Num} \times \mathit{Num})) \\
\mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{id}_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m] \rrbracket (xn, yn) \\
&= (\llbracket \mathit{id}_{\mathcal{P}} [id_1, \dots, id_n] [id'_1, \dots, id'_m] \text{ AT } (xn, yn) \rrbracket, [], (\mathit{across}(id_{\mathcal{P}}), \mathit{upwards}(id_{\mathcal{P}}))) \\
\mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{id}(exp_1, \dots, exp_j) [id_1, \dots, id_n] [id'_1, \dots, id'_m] \rrbracket (xn, yn) \\
&= (\mathit{stmts}', (\mathit{VAR } id_{local_1} : \mathit{WIRE}, \dots, id_{local_p} : \mathit{WIRE}) \# \mathit{locals}', (\mathit{acc}, \mathit{up})) \\
&\quad \text{where } (\mathit{BLOCK } id (gid_1, \dots, gid_j) \\
&\quad\quad [id_{in_1} : \mathit{WIRE}, \dots, id_{in_n} : \mathit{WIRE}] [id_{out_1} : \mathit{WIRE}, \dots, id_{out_m} : \mathit{WIRE}] \\
&\quad\quad \mathit{VAR } lid_1, \dots, lid_q; \\
&\quad\quad \mathit{VAR } id_{local_1} : \mathit{WIRE}, \dots, id_{local_p} : \mathit{WIRE}; \\
&\quad\quad \mathit{BEGIN} \\
&\quad\quad \mathit{bes} \\
&\quad\quad \mathit{END}) = \alpha(\Gamma_B id) \\
&\quad \mu_1 = \{id_1 \mapsto id_{in_1}, \dots, id_n \mapsto id_{in_n}\} \\
&\quad \mu_2 = \{id'_1 \mapsto id_{out_1}, \dots, id'_m \mapsto id_{out_m}\} \\
&\quad \mu_3 = \{id_{local_1} \mapsto id_{local_1}, \dots, id_{local_p} \mapsto id_{local_p}\} \\
&\quad \sigma' = \{gid_1 \mapsto \mathcal{E}_{\sigma} \llbracket exp_1 \rrbracket, \dots, gid_j \mapsto \mathcal{E}_{\sigma} \llbracket exp_j \rrbracket\} \\
&\quad (\mathit{stmts}', \mathit{locals}', (\mathit{acc}, \mathit{up})) = \mathcal{B}(\Gamma_B, \mu_1 \oplus \mu_2 \oplus \mu_3, \sigma') \llbracket \mathit{bes} \rrbracket (xn, yn) \\
\mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{BESIDE } (\mathit{bes}_1; \dots; \mathit{bes}_n) \rrbracket (xn, yn) \\
&= (\mathit{stmts}_1 \# \dots \# \mathit{stmts}_n, \mathit{locals}_1 \# \dots \# \mathit{locals}_n, (\mathit{acc}_1 + \dots + \mathit{acc}_n, \max(\mathit{up}_1, \dots, \mathit{up}_n))) \\
&\quad \text{where } (\mathit{stmts}_1, \mathit{locals}_1, (\mathit{acc}_1, \mathit{up}_1)) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{bes}_1 \rrbracket (xn, yn) \\
&\quad (\mathit{stmts}_2, \mathit{locals}_2, (\mathit{acc}_2, \mathit{up}_2)) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{bes}_2 \rrbracket (xn + \mathit{acc}_1, yn) \\
&\quad \vdots \\
&\quad (\mathit{stmts}_n, \mathit{locals}_n, (\mathit{acc}_n, \mathit{up}_n)) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{bes}_n \rrbracket (xn + \mathit{acc}_1 + \dots + \mathit{acc}_{n-1}, yn) \\
\mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{BELOW } (\mathit{bes}_1; \dots; \mathit{bes}_n) \rrbracket (xn, yn) \\
&= (\mathit{stmts}_1 \# \dots \# \mathit{stmts}_n, \mathit{locals}_1 \# \dots \# \mathit{locals}_n, (\max(\mathit{acc}_1, \dots, \mathit{acc}_n), \mathit{up}_1 + \dots + \mathit{up}_n)) \\
&\quad \text{where } (\mathit{stmts}_1, \mathit{locals}_1, (\mathit{acc}_1, \mathit{up}_1)) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{bes}_1 \rrbracket (xn, yn) \\
&\quad (\mathit{stmts}_2, \mathit{locals}_2, (\mathit{acc}_2, \mathit{up}_2)) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{bes}_2 \rrbracket (xn, yn + \mathit{up}_1) \\
&\quad \vdots \\
&\quad (\mathit{stmts}_n, \mathit{locals}_n, (\mathit{acc}_n, \mathit{up}_n)) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{bes}_n \rrbracket (xn, yn + \mathit{up}_1 + \dots + \mathit{up}_{n-1}) \\
\mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{BESIDE FOR } id = exp_1 .. exp_2 \mathit{ BEGIN } \mathit{bes} \mathit{ END} \rrbracket (xn, yn) \\
&= \text{if } \mathcal{E}_{\sigma} \llbracket exp_1 \rrbracket > \llbracket exp_2 \rrbracket \\
&\quad \text{then } ([], [], (0, 0)) \\
&\quad \text{else } (\mathit{stmts}' \# \mathit{stmts}'', \mathit{locals}' \# \mathit{locals}'', (\mathit{acc}' + \mathit{acc}'', \max(\mathit{up}', \mathit{up}''))) \\
&\quad \quad \text{where } (\mathit{stmts}', \mathit{locals}', (\mathit{acc}', \mathit{up}')) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{bes} \rrbracket (xn, yn) \\
&\quad \quad (\mathit{stmts}'', \mathit{locals}'', (\mathit{acc}'', \mathit{up}'')) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{BESIDE FOR } id = (exp_1 + 1) .. exp_2 \\
&\quad \quad \quad \mathit{BEGIN } \mathit{bes} \mathit{ END} \rrbracket (xn + \mathit{acc}', yn) \\
\mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{BELOW FOR } id = exp_1 .. exp_2 \mathit{ BEGIN } \mathit{bes} \mathit{ END} \rrbracket (xn, yn) \\
&= \text{if } \mathcal{E}_{\sigma} \llbracket exp_1 \rrbracket > \llbracket exp_2 \rrbracket \\
&\quad \text{then } ([], [], (0, 0)) \\
&\quad \text{else } (\mathit{stmts}' \# \mathit{stmts}'', \mathit{locals}' \# \mathit{locals}'', (\max(\mathit{acc}', \mathit{acc}''), \mathit{up}' + \mathit{up}'')) \\
&\quad \quad \text{where } (\mathit{stmts}', \mathit{locals}', (\mathit{acc}', \mathit{up}')) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{bes} \rrbracket (xn, yn) \\
&\quad \quad (\mathit{stmts}'', \mathit{locals}'', (\mathit{acc}'', \mathit{up}'')) = \mathcal{B}(\Gamma_B, \mu, \sigma) \llbracket \mathit{BELOW FOR } id = (exp_1 + 1) .. exp_2 \\
&\quad \quad \quad \mathit{BEGIN } \mathit{bes} \mathit{ END} \rrbracket (xn, yn + \mathit{up}')
\end{aligned}$$

Fig. 18. Mapping descriptions with relative placement information to flattened descriptions in which primitive blocks have numerical placement positions

conditionals occur at the end of a BESIDE or BELOW. We pre-process conditional descriptions so that all calls that occur after a GENERATE IF statement are removed. These calls are nested within either a conditional that succeeds or one that fails for the particular guard. Considering our example above we would arrive at the following description:

```

BESIDE ( a ;
  GENERATE IF x=2
    THEN BESIDE (b;c)
  GENERATE IF NOT (x=2) THEN c)

```

In effect we create a tree of possible placement paths so that each conditional branch will contain all possible subsequent gate calls. The recursive descent algorithm that undertakes this conversion is presented in [MLD02].

The support for conditional statements enables us to build a partial evaluator for Pebble. A partial evaluator is an algorithm which, when given a program and some of its input data, produces a residual or specialised program. Running the residual program on the remaining data will yield the same result as running the original program on all of its input data [JGS93].

Our use of the Pebble language is to enable a parametrised style of hardware design [Luk96]. Partial evaluation, even with no static data at all, can often optimise such descriptions. This is because it can propagate constants from blocks where they are defined to those where they are used, and precomputing wherever possible.

However, in the case of our placement descriptions, we seek to exploit the inefficiency introduced when assigning locations to primitive blocks within conditionals. We assume that the size of a conditional statement is the maximum of both the true and false cases. If we know in advance which branch of the conditional will be chosen, then we can not only eliminate the dead code from our circuit description, but also re-apply the \mathcal{P} function to create a more precise layout.

We demonstrate this process by partially evaluating our pattern matcher example when the value of `specialise` is 1. The size of the loop body of the resulting implementation is smaller, reducing the height of the pattern matcher block from:

$$(n, w \times 3)$$

to:

$$(n, w \times 2)$$

where n is the length of the pattern and w is the bit width of the data. The bounding box of the floorplan of the specialised design is 66% of that of the non-specialised design – in other words, the compaction reduces the resources used by 33%.

We have also developed larger examples using the proposed method than the pattern matcher designs considered in this paper. For instance, our work on DES encryption shows that, when implemented on a Xilinx Virtex FPGA, our compaction technique is able to reduce the size of the design by 60% [MLD02].

10. Summary and future work

We have provided a functional specification for a procedure that compiles a description with relative placement information into a version where symbolic information is specified using coordinates. We have also shown how to give semantics to designs with symbolic relative placement and with symbolic explicit placement coordinates, by mapping them to flattened designs containing primitive blocks with numerical placement coordinates. The correctness of the compilation procedure can then be verified with respect to such semantics.

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. The Quartz language, for instance, supports higher-order functions with overloading [PL05]. These features enable a combinator style of development [JoS90] 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 this Polymorphic Pebble [ML01] is 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 Eq. 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 [JoS90]. The proposed extensions of Pebble take us a step closer to providing, for instance, a generic transformation rule [Luk96] which can be used to derive pipelined designs from a non-pipelined

design. This facility will enable speed and energy optimisation, since recent work [WAL04] has shown that, for reconfigurable hardware technology, pipelined designs can run faster or can consume lower energy per operation than non-pipelined designs. Further work will generalise our approach to deal with relational descriptions [GL01].

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 branch-optimised compilation [SL04] and pipeline optimisation [WeL01], can be based. While future studies will establish the extent to which industrial tools can benefit from our approach, we believe that a provably-correct framework will have a profound impact on understanding the scope and effectiveness of hardware synthesis algorithms and their implementation [MLD02].

Current and future work involves extending our approach to support application-specific and architecture-specific partitioning and placement methods, and exploring their use to guide development and optimisation of realistic designs and tools. Moreover, we aim to integrate relative placement information with the compilation of run-time Pebble descriptions to improve performance and reduce size and power consumption [DLu02]. It would also be useful to explore mechanisation of our verification and compiler generation techniques.

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, Miriam Leeser, Oliver Pell, 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 anonymous reviewers of various drafts of this paper for their comments and suggestions. 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

- [AaL94] Aagaard M, Leeser M (1994) PBS: proven Boolean simplification. *IEEE Trans Comput-Aided Des* 13(4)
- [AaL95] Aagaard M, Leeser M (1995) Verifying a logic-synthesis algorithm and implementation: a case study in software verification. *IEEE Trans. Softw Eng* 21(10)
- [ACG92] Attali I, Chazarain J, Gillette S (1992) Incremental evaluation of natural semantics specifications. In: *Proceedings of the 4th international symposium on programming language implementation and logic programming*, LNCS 631, Springer, Berlin Heidelberg New York
- [Con01] Cong J (2001) An interconnect-centric design flow for nanometer technologies. In: *Proceedings of IEEE*, vol. 89, no 4, pp 505–528
- [DLu02] Derbyshire A, Luk W (2002) Compiling run-time parametrisable designs. In: *Proceedings of IEEE international conference on field-programmable technology*, pp 44–51
- [Den00] Dennis LA et al. (2000) The PROSPER toolkit. In: *Proceedings of the 6th international conference on tools and algorithms for the construction and analysis of systems*, LNCS 1785, Springer
- [Des84] Despeyroux T (1984) Executable specification of static semantics. *Semantics of data types*, LNCS 173, Springer, Berlin Heidelberg New York
- [EBK96] Eisenbiegler D, Blumenroehr C, Kumar R (1996) Implementation issues about the embedding of existing high level synthesis algorithms in HOL. *Theorem proving in higher order logics*, LNCS 1125, Springer
- [GL01] Guo S, Luk W (2001) An integrated system for developing regular array designs. *J Syst Arch* 47(3–4):315–337
- [Han94] Hankin C (1994) *Lambda calculus. A guide for computer scientists*, Oxford University Press, USA
- [HBL96] He J, Brown G, Luk W, O'Leary JW (1996) *Deriving two-phase modules for a multi-target hardware compiler. Designing correct circuits*, Springer Electronic Workshop in Computing, Berlin Heidelberg, New York
- [HPB93] He J, Page I, Bowen JP (1993) *Towards a provably correct hardware implementation of occam. Correct hardware design and verification methods*, LNCS 683, Springer, Berlin Heidelberg, New York
- [HHA93] Hoare CAR, He J, Sampaio A (1993) Normal form approach to compiler design. *Acta Informatica* 30:701–739
- [Hut98] Hutton G (1998) Fold and unfold for program semantics. In: *Proceedings of 3 ACM SIGPLAN international conference on functional programming*, pp 280–288
- [JGS93] Jones N, Gomard C, Sestoft P (1993) *Partial Evaluation and Automatic Programme Generation*, Prentice Hall International Series in Computer Science, Englewood cliffs, USA
- [JoS88] Jones G, Sheeran M (1988) Timeless truths about sequential circuits. In: *Tewksbury SK et al. (ed) Concurrent computations: algorithms, architectures and technology*, Plenum Press, USA
- [JoS90] Jones G, Sheeran M (1990) Circuit design in ruby. In: *Staunstrup J (ed) Formal Methods for VLSI design*, North-Holland
- [JøS86] Jørring U, Scherlis W (1986) *Compilers and staging transformations*. In: *Proceedings of ACM symposium on principles of programming languages*, ACM Press, New York

- [KeG99] Kern C, Greenstreet M (1999) Formal verification in hardware design: a survey. In: Proceedings of ACM transactions design automation of electronic systemen, vol. 4, pp 123–193
- [KBE96] Kumar R, Blumenroehr C, Eisenbiegler D (1996) Formal synthesis in circuit design – a classification and survey. Formal methods in computer-aided design, LNCS 1166, Springer, Berlin Heidelberg, New York
- [LuM97] Luk W, McKeever SW (1998) Pebble: a language for parametrised and reconfigurable hardware design. Field-programmable logic and applications, LNCS 1482, Springer, Berlin Heidelberg, New York
- [LJS89] Luk W, Jones G, Sheeran M (1990) Computer-based tools For regular array design. Systolic array processors, IEEE Computer Society Press, USA
- [Luk96] Luk W, et. al. (1996) A framework for developing parametrised FPGA libraries. Field-programmable logic and applications, LNCS 1142, Springer, Berlin Heidelberg New York
- [Luk99] Luk W, et al. (1999) Reconfigurable computing for augmented reality. In: Proceedings of IEEE symposium on field-programmable custom computing machines, IEEE Computer Society Press, USA
- [MaV98] Mansouri N, Vemuri R (1998) 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, Berlin Heidelberg New York
- [PL05] Pell O, Luk W (to appear) Quartz: a framework for correct and efficient reconfigurable design. In: Proceedings of ReConFig05, international conference on reconfigurable computing and FPGAs
- [ML01] McKeever SW, Luk W (2001) A declarative framework for developing parametrised hardware libraries. In: Proceedings of the 8th international conference on electronics, circuits and systems, IEEE, pp 1635–1638
- [MLD02] McKeever S, Luk W, Derbyshire A (2002) Compiling hardware descriptions using relative placement information for parametrised libraries. In: Proceedings of the 4th international conference on formal methods in computer-aided design, LNCS 2517, Springer, Berlin Heidelberg New York
- [NiN92] Nielson HR, Nielson F (1992) Semantics with applications, Wiley New York
- [Pfs99] Pfenning F, Schrmann C (1999) System description: Twelf – a meta-logical framework for deductive systems. In: Proceedings of international conference on automated deduction, LNAI 1632, Springer, Berlin Heidelberg New York
- [SSS00] Sheeran M, Singh S, Stalmarck G (2000) Checking safety properties using induction and a SAT-solver. In: Proceedings of international conference on formal methods in CAD, LNCS 1954, Springer, Berlin Heidelberg New York
- [SLC00] Shirazi N, Luk W, Chung PYK (2000) Framework and tools for run-time reconfigurable designs. IEE proceedings on computer digital technology, vol. 147, no. 3, pp 147–152
- [S00] Singh S (2000) Death of the RLOC?. In: Proceedings of the symposium on field-programmable custom computing machine, IEEE Computer Society, USA
- [SL04] Styles H, Luk W (2004) Exploiting program branch probabilities in hardware compilation. IEEE Trans Comput 53(11):1408–1419
- [SM98] Susanto KW, Melham T (2001) Formally analysed dynamic synthesis of hardware. J Supercomput 19(1):7–22
- [WeL01] Weinhardt M, Luk W (2001) Pipeline vectorization. IEEE Transactions computer-aided design of integrated circuits and systems vol 20, no 2, pp 234–248
- [WAL04] Wilton S, Ang S-S, Luk W (2004) The impact of pipelining on energy per operation in Field-Programmable Gate Arrays. Field-Prog. logic and applications, LNCS 3203, Springer, Berlin Heidelberg New York

Received July 2004

Revised September 2005

Accepted September 2005 by M. J. Butler

Published online 3 November 2005