Functional Design using Behavioural and Structural Components

Richard Sharp
rws26@cl.cam.ac.uk

University of Cambridge Computer Laboratory
William Gates Building
JJ Thomson Avenue
Cambridge CB3 0FD, UK

Abstract

In previous work we have demonstrated how the functional language SAFL can be used as a *behavioural* hardware description language. Other work (such as μFP and Lava) has demonstrated that functional languages are apposite for *structural* hardware description.

One of the strengths of systems such as VHDL and Verilog is their ability to mix structural- and behavioural-level primitives in a single specification. Motivated by this observation, we describe a unified framework in which a stratified functional language is used to specify hardware across different levels of abstraction: Lava-style structural expansion is used to generate acyclic combinatorial circuits; these combinatorial fragments are composed at the SAFL level. We demonstrate the utility of this programming paradigm by means of a realistic case-study. Our tools have been used to specify, simulate and synthesise a DES encryption/decryption circuit. Area-time performance figures are presented.

1 Introduction

Hardware description languages (HDLs) are often categorised according to the level of abstraction they provide. *Behavioural HDLs* focus on algorithmic specification and attempt to abstract as many low-level implementation issues as possible. Most behavioural HDLs support constructs commonly found in high-level programming languages (e.g. assignment, sequencing, conditionals and iteration). In contrast, *Structural HDLs* allow a hardware engineer to describe a circuit by specifying its hardware-level components and their interconnections. The process of automatically translating a Behavioural HDL into a Structural HDL is often referred to as *high-level synthesis*.

Commercially the two most important HDLs are Verilog and VHDL [8, 7]. A contributing factor to the success of these systems is their support for *both* behavioural *and* structural-level design. The ability to combine behavioural and structural primitives in a single specification offers engineers a powerful framework: when the precise low-level details of a component are not critical, behavioural constructs can be used; for components where finer-grained control is required, structural constructs can be used.[1] However, the flip-side is that by supporting multiple levels of abstraction both Verilog and VHDL are very large languages which are difficult to analyse, transform and reason about.

In previous work we have designed SAFL [11], a *behavioural* HDL which supports a functional programming style. An optimising high-level synthesis system has been implemented which compiles SAFL specifications into structural Verilog [8]. (We map the generated Verilog to silicon using commercially available RTL compilers.) Other researchers have demonstrated that functional languages are powerful tools for *structural* hardware specification [19, 14, 3]. In this paper

[1] Note the analogy with embedding assembly code in a higher-level software language.

we present a system which integrates both structural- and behavioural-level hardware design in a pure functional framework. Our technique involves embedding a functional language designed for structural hardware description into SAFL.

The remainder of this paper is structured as follows. After surveying related work (Section 2) we give a brief overview of the SAFL language (Section 3). Our mechanism for embedding Lava-style structural expansion in SAFL is then presented (Section 4). This methodology is demonstrated by means of a realistic case-study in which a fully functional DES encrypter/decrypter is specified (Section 5).

2 Related Work

There is a large body of work on using functional languages to describe hardware at the structural level. Notable systems in this area include μFP [19], HDRE/Hydra [14], Hawk [9] and Lava [3]. The central idea behind each of these systems is to use the powerful features found in existing functional languages (e.g. higher-order functions, polymorphism and lazy evaluation) to build up netlists from simple primitives. These primitives can be given different semantic interpretations allowing, for example, the same specification to be either simulated or translated into a netlist. However, whilst this technique is obviously appealing, there are problems involved in generating netlists for circuits which contain feedback loops. The difficulty is that, in a pure functional language, a cyclic circuit (expressed as a series of mutually recursive equations) naturally evaluates to an infinite tree preventing the netlist translation phase from terminating.

A number of solutions to this problem have been proposed: O'Donnell advocates the explicit tagging of components at the source-level [15]. In this system the programmer is responsible for labelling distinct components of a circuit with unique values. Whilst this allows a pure functional graph traversal algorithm to detect cycles trivially (by maintaining a list of tags which have already been seen) it imposes an extra burden on the programmer and significantly increases potential for manual error (since it is the programmer's job to ensure that distinct components have unique tags). Lava [3] also uses tagging to identify cycles, but employs a *state monad* [20] to generate fresh tags automatically. Although this neatly abstracts the low-level tagging details from the designer, Claessen and Sands argue that the resulting style of programming is "unnatural" and "inconvenient" [5]. In the same paper, Claessen and Sands propose another solution which involves augmenting Haskell (the functional language in which Lava is embedded) with immutable references which support a test for equality. This extension makes graph sharing observable at the source-level but, although it is shown that many useful laws still hold, full equational reasoning is no longer possible—for example, β-reduction no longer preserves equality.

In this paper we present an alternative approach. By only allowing the description of *acyclic* circuits through Lava-style structural static expansion and then combining these circuit fragments at the SAFL level we facilitate the *pure functional* specification of complex circuits which can contain feedback loops. We have not solved the observable sharing problem; instead we have eliminated it: since cycles are not permitted at the structural level we do not have to worry about infinite loops being statically expanded. Conversely, since feedback loops are represented as tail-recursive calls at the SAFL-level there is no need to introduce impure language features.

Although most of the work on using functional languages for hardware description focuses on the *structural* level some researchers have considered using functional languages for *behavioural* hardware description. Johnson's Digital Design Derivation (DDD) system [4] uses a scheme-like language to describe circuit behaviour. A series of semantics-preserving transformations are presented which can be used to refine a behavioural specification into a circuit structure; the transformations are applied manually by an engineer. This is a different approach to hardware design using SAFL [11] where, although semantics-preserving transformations are used to explore architectural tradeoffs (including allocation, binding and scheduling [6]) at the source-level, the resulting SAFL specification is fed into an optimising compiler which generates a structural hardware design automatically.

3 Overview of the SAFL Language

SAFL has syntactic categories e (term) and p (program). First suppose that v ranges over a set of constants. Let x range over variables (occurring in `let` declarations or as formal parameters), a over primitive functions (such as addition) and f over user-defined functions. For typographical convenience we abbreviate formal parameter lists $(x_1, \ldots, x_k)$ and actual parameter lists $(e_1, \ldots, e_k)$ to $\vec{x}$ and $\vec{e}$ respectively; the same abbreviations are used in `let` definitions. Then the abstract syntax of the core SAFL language can be given in terms of recursion equations over programs, p, and expressions, e:

$$\begin{array}{rcl} e & ::= & v \mid x \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0 \mid \\ & & a(e_1, \ldots, e_{arity(a)}) \mid f(e_1, \ldots, e_{arity(f)}) \\ p & ::= & \texttt{fun } f_1(\vec{x}) = e_1 \quad \ldots \quad \texttt{fun } f_n(\vec{x}) = e_n \end{array}$$

It is sometimes convenient to extend this syntax slightly. In later examples we use a `case`-expression instead of iterated tests; we also write `e[n:m]` to select a bit-field `[n..m]` from the result of expression `e` (where `n` and `m` are integer constants).

There is a syntactic restriction that whenever a call to function f_j from function f_i is part of a cycle in the call graph of p then we require the call to be a tail call.[2] (Note that calls to a function not forming part of a cycle can occur in an arbitrary expression context.) This ensures that storage for the variables and temporaries of p can be allocated statically—in software terms the storage is associated with the code of the compiled function; in hardware terms it is associated with the logic to evaluate the function body.

The other main feature of SAFL, apart from static allocatability, is that its evaluation is limited only by data flow (and control flow at user-defined call and conditional). Thus, in the form $\texttt{let } \vec{x} = (e_1, \ldots, e_k) \texttt{ in } e_0$ or in a call $f(e_1, \ldots, e_k)$ or $a(e_1, \ldots, e_k)$, all the e_i $(1 \leq i \leq k)$ are evaluated concurrently. In the conditional $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$ we first evaluate (only) e_1; one of e_2 or e_3 is evaluated after its result is known. SAFL has call-by-value semantics since eager evaluation offers a greater opportunity for parallelism (i.e. we can execute a function call's arguments in parallel without worrying about strictness).

Although up to this point we have referred to SAFL as a behavioural language, it is also capable of capturing some structural aspects of a design. We say that SAFL is *resource-aware* to indicate that a single user-defined function definition at the source-level corresponds to a single hardware resource at the circuit-level. In this context multiple calls to the same function corresponds to resource sharing[3]. We use SAFL-level transformations to express architectural tradeoffs such as resource duplication/sharing and hardware/software co-design [12]. In essence these transformations preserve a specification's *extensional* semantics (the result returned) whilst changing the *intensional* semantics (how the circuit is structured). A more in-depth description of the SAFL language and its associated silicon compiler can be found in our recent survey paper [13]. For the purposes of this document we provide a short example which illustrates the main points:

```
fun mult(x:16, y:16, acc:32):32 =
  if (x=0 | y=0) then acc
     else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)

fun f(x:16):32 = mult(x, x, 0) + mult(13, x, 0)
```

From this specification, two hardware resources are generated: a circuit, $H_{\texttt{mult}}$, corresponding to `mult` and a circuit, $H_{\texttt{f}}$, corresponding to `f`. The two calls to `mult` are not inlined: at the hardware level there is only one shared resource, $H_{\texttt{mult}}$, which is invoked twice by $H_{\texttt{cube}}$. The tail-recursive call in the definition of `mult` is synthesised into a feedback loop at the circuit level. Since function

[2] Tail calls consist of calls forming the whole of a function body, or nested solely within the bodies of `let-in` expressions or that are the consequents of `if-then-else` expressions.

[3] Our optimising compiler automatically deals with sharing issues by statically scheduling access to resources where it can, and generating arbiters to perform scheduling dynamically otherwise [18].

arguments are evaluated concurrently, the two shift operations occurring in the recursive call to `mult` are evaluated in parallel along with the conditional test and possibly, depending on the conditional branch taken, the addition operation.

Each SAFL variable is annotated with a bit-width at its point of introduction. We use the form `x:`w to indicate that variable `x` has width w. Note that the widths of function result types are also specified explicitly (using the form `fun f(...):`w). Widths of constants can either be specified explicitly or, more usually, inferred from their local context. As part of a simple type-checking phase our SAFL compiler ensures that for each function call, $f(\vec{x})$, the widths of arguments, $\vec{x}$ match those specified in the signature of f.

4 Embedding Structural Expansion in SAFL

Resource awareness allows SAFL to describe the *system-level* structure of a design by mapping `fun` declarations to circuit-level functional units. In contrast, systems such as μFP and Lava offer much finer-grained control over circuit structure, taking logic-gates (rather than function definitions) as their structural primitives. We are not arguing that either approach is better: in practice both are appropriate depending on the type of hardware that is being designed. Motivated by this observation, we present a framework which integrates Lava-style structural expansion with SAFL.

Section 4.1 outlines our system for fine-grained structural hardware description which, for the purposes of this paper, we will refer to as Magma[4]. In Section 4.2 we show how Magma is integrated with SAFL.

4.1 Building Combinatorial Hardware in Magma

An argument in favour of Lava, Hydra and other similar systems, is that since they are embedded in existing functional languages they are able to leverage existing tools and compilers. Furthermore, use of non-standard interpretation of basis functions means that the *same* compiler can be used to perform both hardware simulation and synthesis. These compelling benefits lead us to adopt a similar approach. However, in contrast to Lava, which is embedded in Haskell [1], we choose to embed Magma in ML [10]. The choice of ML is fitting for two main reasons: firstly, since we only wish to describe acyclic circuits, ML's strict evaluation is appropriate for both simulation and synthesis interpretations; secondly, since SAFL also borrows much of its syntax and semantics from ML, both Magma and SAFL share similar conventions (an important consideration when we are dealing with specifications containing a mixture of both Magma and SAFL).

4.1.1 An ML module system primer

In order to understand the workings of Magma some familiarity with the ML module system is required. Whilst we do not describe the full details of the ML-module system here, this section is sufficient to allow readers unfamiliar the module system to understand the remainder of this paper. For more information the reader is referred to a more in-depth survey [16].

The basic element of ML's module system is the `structure`. The structure provides a way of packaging both type and value (including function) definitions into a single entity. An important feature of structures is that they provide a hierarchical name-space. For example, if a function, f, is defined in a structure $\mathcal{S}$ we refer to it as $\mathcal{S}.f$.

An ML `signature` provides a mechanism to specify interfaces. A signature contains a set of name and type-declarations. One can use a signature to constrain a structure using the "`:`" operator. Only values whose types are explicitly declared in the constraining signature are visible outside the constrained structure.

Finally, the ML module system provides `functor`s. A functor is essentially a parameterised structure, dependent on another structure which is provided externally. For example, consider the

[4] As it is a restricted form of Lava.

```
signature BASIS =
    sig
        type bit
        val b0   : bit
        val b1   : bit
        val orb  : bit * bit -> bit
        val andb : bit * bit -> bit
        val notb : bit * bit
        val xorb : bit * bit -> bit
    end
```

Figure 1: The definition of the BASIS signature (from the Magma library)

following (contrived) code fragment which defines a functor, FTR, parameterised over a structure, S (where S is constrained by signature, SSIG):

```
functor FTR(S:SSIG) =
  struct
     val a = S.f(3)
  end
```

Passing a structure, T, into FTR yields a new structure containing a single item, a, which has the value T.f(3). Magma makes use of functors to parameterise hardware specifications over interpretations of their basis functions. This provides a convenient way of using the same code for both simulation and synthesis (see below).

4.1.2 Specifying Hardware in Magma

The Magma system essentially consists of a library of ML code. A signature called BASIS is provided which declares the types of supported basis functions (see Figure 1). Values b0 and b1 correspond to logic-0 (false) and logic-1 (true) respectively. Functions orb, andb, notb and xorb correspond to logic functions *or*, *and*, *not* and *xor*. Two structures which implement BASIS are provided:

- SimulateBasis provides a simulation interpretation. We implement bits as boolean values; functions orb, andb etc. have their usual boolean interpretations.
- SynthesisBasis provides a synthesis interpretation. We implement bits as strings representing names of wires in a net-list. Functions orb, andb etc. take input wires as arguments and return a (fresh) output wire. Calling one of the basis functions results in its netlist declaration being written to the selected output stream as a side-effect. For example, if the result of calling andb with string arguments "in_wire1" and "in_wire2" is the string "out_wire" then the following is output to StdOut:

```
and(out_wire,in_wire1,in_wire2);
```

Figure 2 shows a Magma specification of a ripple-adder. As with all Magma programs, the main body of code is contained within an ML functor. This provides a convenient abstraction, allowing us to parameterise a design over its basis functions. By passing in the structure SimulateBasis (see above) we are able to instantiate a copy of the design for simulation purposes; similarly, by passing in SynthesisBasis we instantiate a version of the design which, when executed, outputs its netlist. The signature RP_ADD is used to specify the type of the ripple_add function. Using this signature to constrain the RippleAdder functor also means that *only* the ripple_add function is externally visible; the functions carry_chain and adder can only be accessed from within the

```
signature RP_ADD =
  sig
      type bit
      val ripple_add : (bit list * bit list) -> bit list
  end

functor RippleAdder (B:BASIS):RP_ADD =
      struct

         type bit=B.bit
         fun adder (x,y,c_in) = (B.xorb(c_in, B.xorb(x,y)),
                                 B.orb( B.orb( B.andb (x,y), B.andb(x,c_in)),
                                        B.andb(y,c_in)))

         fun carry_chain f _ ([],[]) = []
           | carry_chain f c_in (x::xs,y::ys) =
                 let val (res_bit, c_out) = f (x,y,c_in)
                  in res_bit::(carry_chain f c_out (xs,ys))
                 end

         val ripple_add = carry_chain adder B.b0
      end
```

Figure 2: A simple ripple-adder described in Magma

functor. Note that the use of signatures to specify interfaces in this way is not compulsory but, for the usual software-engineering reasons, it is recommended.

Let us imagine that a designer has just written the ripple-adder specification shown in Figure 2 and now wants to test it. This can be done by instantiating a simulation version of the design in an interactive ML session:

```
- structure SimulateAdder = RippleAdder (SimulationBasis);
```

The adder can now be tested by passing in arguments (a tuple of `bit` lists) and examining the result. For example:

```
- SimulateAdder.ripple_add ([b1,b0,b0,b1,b1,b1],[b0,b1,b1,b0,b1,b1])
val it = [b1,b1,b1,b1,b0,b1]  : SimulateAdder.bit list
```

Let us now imagine that the net-list corresponding to the rippler-adder is required. We start by instantiating a synthesis version of the design:

```
- structure SynthesiseAdder = RippleAdder (SynthesisBasis);
```

If we pass in lists of input wires as arguments, the `ripple_add` function prints its netlist to the screen and returns a list of output wires:

```
- SynthesiseAdder.ripple_add (Magma.new_bus 5, Magma.new_bus 5)
and(w_1,w_45,w_46);
and(w_2,w_1,w_44);
...
and(w_149,w_55,w_103);
val it = ["w_149","w_150","w_151","w_152","w_153"]
```

The function `new_bus`, part of the Magma library, is used to generate a bus of given width (represented as a list of wires).

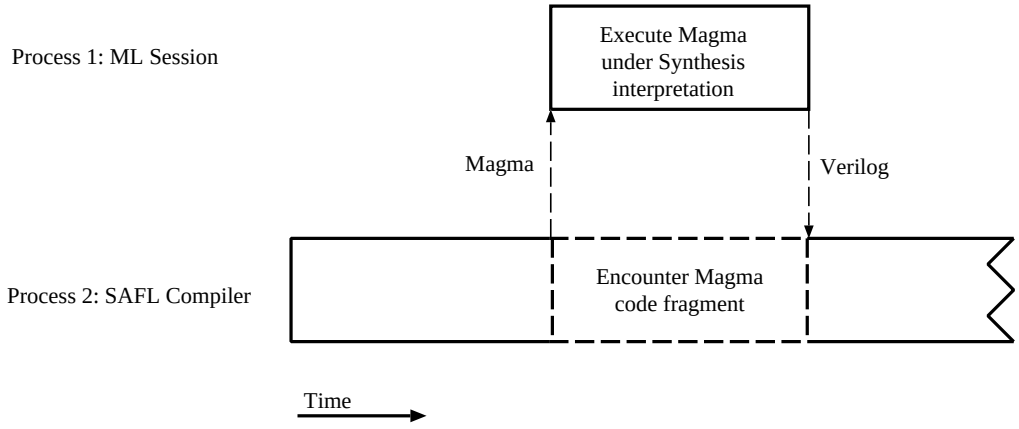

Figure 3: A diagrammatic view of the steps involved in compiling a SAFL/Magma specification

```
(* Magma library block containing Magma_Code functor: *)

<%
  signature RP_ADD =
     ... (* as in Figure 2 *)

  functor Magma_Code (B:BASIS):RP_ADD =
     ... (* as RippleAdder functor in Figure 2 *)
%>

fun mult(x, y, acc) =
  if (x=0 | y=0) then acc
    else mult(x<<1, y>>1,
              if y[0] then <% ripple_add %>(acc,x) else acc)
```

Figure 4: A simple example of integrating Magma and SAFL into a single specification

4.2 Integrating SAFL and Magma

Our approach to integrating Magma and SAFL involves using delimiters to embed Magma code fragments inside SAFL programs. At compile time the embedded Magma is synthesised and the resulting netlist is incorporated into the generated circuit (see Figure 3). This technique was partly inspired by web-scripting frameworks such as ASP and PHP [2] which can be embedded in HTML documents[5]. To highlight this analogy we use ASP-style delimiters "`<%`" and "`%>`" to mark the start and end points of Magma code fragments. Our compiler performs simple width checking across the SAFL-Magma boundary, ensuring the validity of the final design.

The SAFL parser is extended to allow a special type of Magma code fragment at the beginning of a specification. This initial Magma fragment, which is referred to as the *library block*, contains an ML functor called `Magma_Code`. Functions within `Magma_Code` can be called from other Magma fragments in the remainder of the specification. Figure 4 illustrates these points with a simple example in which the Magma ripple adder (initially defined in Figure 2) is invoked from a SAFL specification. The precise details of the SAFL-Magma integration are discussed later in this section; for now it suffices to observe that Magma fragments are treated as functions at the SAFL-level and applied to SAFL expressions.

[5]When a dynamic web-page is fetched the ASP or PHP code is executed generating HTML which is returned to the client.

The treatment of Magma fragments is similar to that of primitive functions (such as `+`, `-`, `*` etc.). In particular, Magma code fragments are expanded in-place. For example, if a specification contains two Magma fragments of the form, `<% ripple_add %>`, then the generated hardware contains two separate ripple adders. Note that if we require a shared `ripple_adder` then we can encapsulate the Magma fragment in a SAFL function definition and rely on SAFL's resource-awareness properties. For example, the specification:

```
fun add(x, y) = <% ripple_add %> (x,y)
fun mult_3(x) = add(x, add(x,x))
```

contains a single ripple adder shared between the two invocations within the definition of the `mult_3(x)` function. Since embedded Magma code fragments represent pure functions (i.e. do not cause side effects) they do not inhibit SAFL-level program transformation. Thus our existing SAFL-level transformations corresponding to resource duplication/sharing [11], hardware/software co-design [12] etc. remain valid.

Implementation and Technical Details

Consider the general case of a Magma fragment, m, embedded in SAFL:

$$\texttt{<\%}\ m\ \texttt{\%>}(e_1, \ldots, e_k)$$

where $e_1, \ldots, e_k$ are SAFL expressions. On encountering the embedded Magma code fragment, `<%` m `%>`, our compiler performs the following operations:

1. An ML program, $\mathcal{M}$, (represented as a string) is constructed by concatenating the *library block* together with commands to instantiate the `Magma_Code` functor in its synthesis interpretation (see above).
2. The bit-widths of SAFL expressions, $e_1, \ldots, e_k$, are determined (bit-widths of variables are known to the SAFL compiler) and ML code is added to $\mathcal{M}$ to construct corresponding busses, $B_1, \ldots, B_k$, of the appropriate widths (using the `Magma.new_bus` library call).
3. $\mathcal{M}$ is further augmented with code to:
   (a) execute ML expression, $m(B_1, \ldots, B_k)$, which, since the library block has been instantiated in its synthesis interpretation, results in the generation of a netlist; and
   (b) wrap up the resulting netlist in a Verilog `module` declaration (adding Verilog `wire` declarations as appropriate).
4. A new ML session is spawned as a separate process and program $\mathcal{M}$ is executed within it.
5. The output of $\mathcal{M}$, a Verilog module declaration representing the compiled Magma code fragment, is returned to the SAFL compiler where it is added to the object code. Our SAFL compiler also generates code to instantiate the module, connecting it to the wires corresponding to the output ports of SAFL expressions $e_1, \ldots, e_k$.

In order that the ML-expression $m(B_1, \ldots, B_k)$ type checks, m must evaluate to a function, $\mathcal{F}$, with a type of the form:

```
(bit list * bit list * ... * bit list)  ->  bit list
```

with the arity of $\mathcal{F}$'s argument tuple equal to k. If m does not have the right type then a type-error is generated in the ML-session spawned to execute $\mathcal{M}$. Our SAFL compiler traps this ML type-error and generates a meaningful error of its own, indicating the offending line-number of the SAFL/Magma specification. In this way we ensure that the bit-widths and number of arguments applied to `<%` m `%>` at the SAFL-level match those expected at the Magma-level.

Another property we wish to ensure at compile time is that the output port of a Magma-generated circuit is of the right width. We achieve this by incorporating width information corresponding to the output port of Magma-generated hardware into our SAFL compiler's type-checking phase. Determining the width of a Magma specification's output port is trivial—it is simply the length of the `bit` list returned when $m(B_1, \ldots, B_k)$ is executed.

5 Case Study: DES Encrypter/Decrypter

Appendix A presents code fragments from the SAFL specification of a Data Encryption Standard (DES) encryption/decryption circuit. Here we describe the code for the DES example, focusing on the interaction between SAFL and Magma; the details of the DES algorithms are not discussed. We refer readers who are interested in knowing more about DES to Scheier's cryptography textbook [17].

The library block at the beginning of the DES specification defines three functions used later in the specification:

- `perm` is a curried function which takes a permutation pattern, p, (represented as a list of integers) and a list of bits, l. It returns l permuted according to pattern p.
- `ror` is a curried function which takes an integer, x, and a list of bits, l. It returns l rotated right by x.
- `rol` is as `ror` but rotates bits left (as opposed to right).

A set of permutation patterns required by the DES algorithm are also declared. (For space reasons the bodies of some of these declarations are omitted.)

The code in Appendix A uses two of SAFL's features which have not been described in this paper:

- The primitive function `join` takes an arbitrary number of arguments and returns the bit-level concatenation of these arguments. As one would expect, the bit-width of the result of a call to `join` is the sum of the bit-widths of its input arguments.
- SAFL's `type` declaration allows us to construct records with named fields. Curly braces, { ... }, are used as record constructors and dot notation ($r.f$) is used to select a field, f, from record r. After type-checking our SAFL compiler translates record notation directly into bit-level `join`s and selects. (Recall that bit-level selects are represented using the `e[n:m]` notation—see Section 3.)

Primitive functions corresponding to arithmetic and boolean operators use their standard symbols (e.g. `+`, `<`, `=`). The binary infix operator, (`^`), is used for bit-wise exclusive-or.

The DES algorithm requires 8 S-boxes, each of which is a substitution function which takes a 6-bit input and returns a 4-bit output. The S-boxes' definitions make use of one of SAFL's syntactic sugarings:

`lookup` e `with` {v_0`, ...,` v_k}

Semantically the `lookup` construct is equivalent to a `case` expression:

`case` e `of 0 =>` v_0 `| ...|` $(k-1)$ `=>` v_{k-1} `default` v_k.

To ensure that each input value to the `lookup` expression has a corresponding output value we enforce the constraint that $k = 2^w - 1$ where w is the width of expression e. Our compiler is often able to map `lookup` statements directly into ROM blocks, leading to a significantly more efficient implementation than a series of iterated tests.

Before applying its substition each S-box permutes its input. We use our Magma permutation function to represent this permutation: `<% perm p_inSbox %>(x)`. Other examples of SAFL-Magma integration can be seen throughout the specification. The `keyshift` function makes use

of the Magma `ror` and `rol` functions to generate a key schedule. Other invocations of the Magma `perm` function can be seen in the bodies of SAFL-level functions: `round` and `main`. We find the use of higher-order Magma functions (such as `perm`, `ror` and `rol`) to be a powerful idiom.

We used our tools to map the DES specification to synthesisable RTL-Verilog. A commercial RTL-synthesis tool (Leonardo from Exemplar) was used to synthesise the RTL-Verilog for an Altera Apex E20K200E FPGA (200K gate equivalent). The resulting circuit utilised 8% of the FPGA's resources and could be clocked up to 48MHz. The design was mapped onto a Altera Excalibur Development Board and, using the board's 33MHz reference clock a throughput of 15.8Mb/sec was achieved.

6 Conclusions and Further Work

In this paper we have motivated and described a technique for combining both behavioural and structural-level hardware specification in a stratified pure functional language. Our methodology has been applied to a realistic example. We believe that the major advantages of our approch are as follows:

- As in Verilog and VHDL, we are able to describe large systems consisting of both behavioural and structural components.
- SAFL-level program transformation remains a powerful technique for architectural exploration. The functional nature of the Magma-integration means that our existing library of SAFL transformations are still applicable.
- By only dealing with combinatorial circuits at the structural-level we eliminate the problems associated with graph-sharing in a pure functional language (see Section 2). We do not sacrifice expressivity: cyclic (sequential) circuits can still be formed by composing combinatorial fragments at the SAFL-level in a more controlled way.

In future work we would like to investigate using similar techniques for integrating systems such as Lava or Magma directly into Verilog. Whilst this would not give the formal benefits of our pure functional SAFL-Magma hybrid, it may help to make the tried-and-tested technique of embedding a structural hardware specification using functional languages more accessible to engineers working in industry.

Acknowledgements

This work was supported by (UK) EPSRC grant, reference GR/N64256: "A Resource-Aware Functional Language for Hardware Synthesis"; AT&T Research Laboratories Cambridge provided additional support (including sponsoring the author). The author would like to thank Alan Mycroft for his valuable comments and suggestions.

Appendix A: SAFL specification of a DES encrypter/decrypter

```
(* Magma Library block *)

<%
signature DES =
  sig
     val perm: int list -> 'a list -> 'a list
     val ror:  int -> 'a list -> 'a list
     val rol:  int -> 'a list -> 'a list

     val p_compress : int list
     val p_key      : int list
      ...
     val p_inSbox   : int list
  end

functor Magma_code (B:BASIS):DES =
    struct

      (* DES permutation patterns ... *)

        val p_initial   = [58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
                           62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
                           57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
                           61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7]

        val p_key       = [57,49,41,33,25,17,9,1,58,50,42,34,26,18,
                           10,2,59,51,43,35,27,19,11,3,60,52,44,36,
                           63,55,47,39,31,23,15,7,62,54,46,38,30,22,
                           14,6,61,53,45,37,29,21,13,5,28,20,12,4]

        val p_pbox      = [16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,
                           2,8,24,14,32,27,3,9,19,13,30,6,22,11,4,25]
        val p_compress  = [ ... <snip> ... ]
        val p_expansion = [ ... <snip> ... ]
        val p_final     = [ ... <snip> ... ]
        val p_inSbox    = [1,5,2,3,4]

     (* Higher-order permutation function -- given a list of bits
          and a pattern it returns a permuted list of bits: *)

        fun perm positions input =
            let val inlength = length input
                fun do_perm [] _ = []
                  | do_perm (p::ps) input =
                     (List.nth (input,inlength-p))::(do_perm ps input)
            in do_perm positions input
            end

      (* Rotate bits right by specified amount: *)
        fun ror n l =
            let val last_n = rev (List.take (rev l, n))
```

```
                val rest   = List.take (l, (length l)-n)
             in last_n @ rest
            end

      (* Rotate bits left by specified amount: *)

        fun rol n l =
            let val first_n = List.take (l, n)
                val rest    = List.drop (l, n)
            in rest @ first_n
            end

    end
%>

(* Definitions of S-Boxes (implemented as simple lookup tables) *)

fun sbox1(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,
                 0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
                 4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
                 15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}

fun sbox2(x:6):4 =
  lookup <% perm p_inSbox %> (x)
           with { ... <snip> ... }

fun sbox3(x:6):4 = ...
      ...
fun sbox8(x:6):4 = ...

(* Do s_box substitution on data-block: *)

fun s_sub(x:48):32 =
  join( sbox1( x[47:42] ), sbox2( x[41:36] ),
        sbox3( x[35:30] ), sbox4( x[29:24] ),
        sbox5( x[23:18] ), sbox6( x[17:12] ),
        sbox7( x[11:6]  ), sbox8( x[5:0]   ))

(* Define a record which contains the left and right halves
   of a 64-bit DES block and the 56-bit key. *)

type round_data = record {left:32, right:32, key:56}

(*  Successive keys are calculated by circular shifts. The degree
    of the shift depends on the round (rd).
    We shift either left/right depending on whether we are
    decrypting/encrypting. Note that the inline pragma ensures that
    keyshift is never treated as a shared resource. *)

inline fun keyshift(key_half:28,rd:4,encrypt:1):28 =
  define val shift_one = (rd=0 or rd=1 or rd=8 or rd=15)
  in
```

```
     if encrypt then
         if shift_one then <% rol 1 %> (key_half)
                      else <% rol 2 %> (key_half)
     else
         if rd=0 then key_half
                 else if shift_one then <% ror 1 %> (key_half)
                                   else <% ror 2 %> (key_half)
  end

(*  A single DES round:  *)

inline fun round(bl:round_data,rd:4,encrypt:1):round_data =
    let
         val lkey = keyshift(slice(bl.key,55,28),rd,encrypt)
         val rkey = keyshift(slice(bl.key,27,0),rd,encrypt)
         val keybits   = <% perm p_compress %> ( join(lkey,rkey) )
         val new_right = let val after_p = <% perm p_expansion %>(bl.right)
                         in s_sub (after_p ^ keybits ^ bl.left)
                         end

      in {left=bl.right, right=new_right, key=join(lkey,rkey)}
     end

(*  Do 16 DES rounds: *)

fun des(c:4, rd:round_data,encrypt:1):round_data =
  let
         val new_data = round(rd, c, encrypt)
  in if c=15 then new_data
             else des(c+1, new_data,encrypt)
  end

(*  Apply Key-Permutation to incoming 64 key bits,
    apply the initial permutation to incoming data-block,
    do 16-rounds of DES on permuted data-block,
    apply the final-permutation and return the encrypted block *)

fun main(block:64,key:64, encrypt:1):64 =
  let
         val block_p = <% perm p_initial %> (block)
         val realkey = <% perm p_key %> (key)
         val output = des(0:4, {left=slice(block_p,63,32),
                                right=slice(block_p,31,0),
                                key=realkey}, encrypt)

      in <% perm final %> (join(output.right, output.left))
     end
```

References

[1] Haskell98 report. Available from `http://www.haskell.org/`.

[2] PHP hypertext preprocessor. See `http://www.php.net/`.

[3] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: Hardware description in Haskell. In *Proceedings of the 3rd International Conference on Functional Programming* (1998), SIGPLAN, ACM.

[4] Bose, B. DDD: A transformation system for digital design derivation. Tech. Rep. 331, Indiana University, 1991.

[5] Claessen, K., and Sands, D. Observable sharing for functional circuit description. In *Asian Computing Science Conference* (1999), pp. 62–73.

[6] De Micheli, G. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Inc., 1994.

[7] IEEE. Standard VHDL Reference Manual, 1993. IEEE Standard 1076-1993.

[8] IEEE. Verilog HDL language reference manual. IEEE Draft Standard 1364, October 1995.

[9] Matthews, J., Cook, B., and Launchbury, J. Microprocessor specification in Hawk. In *Proceedings of the IEEE International Conference on Computer Languages* (1998).

[10] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[11] Mycroft, A., and Sharp, R. A statically allocated parallel functional language. In *Proceedings of the International Conference on Automata, Languages and Programming* (2000), vol. 1853 of *LNCS*, Springer-Verlag.

[12] Mycroft, A., and Sharp, R. Hardware/software co-design using functional languages. In *Proceedings of TACAS* (2001), vol. 2031 of *LNCS*, Springer-Verlag.

[13] Mycroft, A., and Sharp, R. Higher-level techniques for hardware description and synthesis. *To appear. International Journal on Software Tools for Technology Transfer (STTT)* (2002).

[14] O'Donnell, J. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications* (April 1987), North-Holland, pp. 363–382.

[15] O'Donnell, J. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming, Workshops in Computing, Proceedings* (1992), Springer-Verlag, pp. 178–194.

[16] Paulson, L. *ML for the working programmer*. Cambridge University Press, 1996.

[17] Schneier, B. *Applied cryptography: protocols, algorithms, and sourcecode in C*. John Wiley and Sons, New York, 1994.

[18] Sharp, R., and Mycroft, A. Soft scheduling for hardware. In *Proceedings of the 8th International Static Analysis Symposium* (2001), vol. 2126 of *LNCS*, Springer-Verlag.

[19] Sheeran, M. muFP, a language for VLSI design. In *Proceedings of the ACM Symposium on LISP and Functional Programming* (1984).

[20] Wadler, P. Monads for functional programming. In *Advanced Functional Programming* (1995), vol. 925 of *LNCS*, Springer-Verlag.