Semantics Driven Hardware Design, Implementation, and Verification with ReWire

Adam Procter William L. Harrison
Ian Graves Michela Becchi
University of Missouri
procteram@missouri.edu
harrisonwl@missouri.edu
iangraves@mail.missouri.edu
becchim@missouri.edu

Gerard Allwein
U.S. Naval Research Laboratory
gerard.allwein@nrl.navy.mil

Abstract

There is no such thing as high assurance without high assurance hardware. High assurance hardware is essential, because any and all high assurance systems ultimately depend on hardware that conforms to, and does not undermine, critical system properties and invariants. And yet, high assurance hardware development is stymied by the conceptual gap between formal methods and hardware description languages used by engineers. This paper presents ReWire, a functional programming language providing a suitable foundation for formal verification of hardware designs, and a compiler for that language that translates high-level, semantics-driven designs directly into working hardware. ReWire's design and implementation are presented, along with a case study in the design of a secure multicore processor, demonstrating both ReWire's expressiveness as a programming language and its power as a framework for formal, high-level reasoning about hardware systems.

1. Introduction

This paper presents ReWire, a high level functional language for designing, implementing and verifying high assurance hardware systems. ReWire is a formally defined programming language for expressing reactive, concurrent, and parallel computations. ReWire is a computational λ-calculus [22] and, as such, is conducive to formal verification [14] of, in particular, security and safety properties [15, 17]. ReWire is a subset of Haskell, where the subset has been carefully chosen so that every ReWire program may be compiled to working hardware implementations. In this work, we outline the design of the ReWire language, and present a substantial case study demonstrating that ReWire supports the rapid development and implementation of provably secure hardware.

The philosophy driving the development of ReWire is the conviction that *semantic archaeology is the bane of high assurance computing*. By "semantic archaeology", we mean the process of developing a formal specification for an *existing* computing artifact. Semantic archaeology is time-consuming and expensive, because such artifacts are rarely written with formal semantics in mind, and, consequently, the formal methods scientist must attempt a painstaking reconstruction of the system semantics from informal and often incomplete natural language documents (if, indeed, such a reconstruction is even possible). ReWire aims to eliminate the need for semantic archaeology by encouraging a "semantics-first" design style. This is achieved by providing (1) a source language with a clear semantic foundation, (2) a compiler that translates these source language programs directly into working implementations, and (3) a formal framework for expressing and proving desired properties (including, but not limited to, security).

The semantics-aware design process supported by ReWire is especially valuable when semantically novel features are important, as is increasingly the case with modern architectures. Consider, for example, hardware-level support for the enforcement of secure information flow policies. Traditionally, information flow concerns are handled in software by a separation kernel [28], which ensures that the sharing of hardware resources by high- and low-security processes does not result in unauthorized information flow. For embedded systems, however, the overhead of such a software-based solution may be cost prohibitive. Furthermore, moving this separation functionality at least partially into hardware [23, 33] can bring great benefit with respect to reliability and verifiability; in high assurance applications, this fact alone may justify the expenditure of a significant fraction of one's transistor budget.

The technical burden of constructing and verifying a separating processor is, however, quite substantial. Section 4 presents a case study demonstrating that ReWire substantially lightens that burden. Beginning with a ReWire-based implementation of a single-core processor supporting an off-the-shelf instruction set architecture, we will see that the extension of this design to a secure, verified dual-core processor may be undertaken with no instrumentation or modification of the original design, and that the proof of security is concise and readable.

The point of departure for this work is the application of ideas from monadic semantics (esp., monads of resumptions and state and effect types) to the modeling and verification of concurrent systems [15–17]. As design tools, these ideas have many virtues. They are flexible and expressive, support formal analysis, and can be readily simulated with any Haskell implementation. In this paper we demonstrate that modular monadic semantics, previously applied to software artifacts such as interpreters and compilers [21] and operating system kernels [4, 15], is also useful in the realm of hardware design. Previous work [26] presents an informal description of ReWire's design in the form of a short paper. This paper

LCTES'15, June 18–19, 2015, Portland, OR, USA.

ACM 978-1-4503-3257-6//...$15.00.
http://dx.doi.org/10.1145/2670529.2754970

provides a substantial technical discussion of the design and implementation of ReWire not to be found in the earlier work.

In summary, the key contributions of this paper are as follows. (1) A novel, semantics-driven, modular style of hardware specification. We show that, in contrast with traditional design techniques typified by mainstream hardware design languages like VHDL, semantics-driven designs may easily be extended with new semantic features without the need to rearchitect large portions of the design. (2) Techniques and tools (i.e., the ReWire compiler) for synthesizing hardware circuits directly from these specifications. (3) A semantics-guided approach to hardware verification wherein separate semantic features may be reasoned about independently, thus reducing the complexity of formal verification both for new designs and for existing designs extended with new features.

The remainder of the paper proceeds as follows. The design of the ReWire language is discussed in detail in Sec. 2. The implementation strategies taken by the ReWire compiler are presented in Sec. 3. Sec. 4 discusses the implementation and verification of a secure processor in ReWire. Finally, Sec. 5 presents conclusions and outlines related and future work.

2. Design of ReWire

The approach to semantics-driven hardware design advocated here centers on a computational λ-calculus and programming language called ReWire, as well as the ReWire compiler which implements this calculus. (We often refer to both the language and the compiler as "ReWire" for short.) As a programming language, ReWire forms a subset of Haskell, including support for a certain class of monads called *reactive resumption monads* which embody the semantic essence of clocked, sequential, reactive computation. Subsetting Haskell has two major advantages: first, existing Haskell programming environments and tools may be used for simulating and testing ReWire designs in software, as ReWire designs are simply computations in a particular monad. Second, during the initial design stage one may utilize the full range of Haskell features—higher order functions, recursive algebraic data types, and so on—to produce a high-level specification, then use semantics-preserving source-to-source program transformations (either by hand or automatically) to produce an implementable circuit specification in the ReWire subset. Previous work [24] has seen the development of a complete, formal denotational semantics of the ReWire language independent of its embedding in Haskell, including its type system and a complete specification of its restrictions on recursion.

The subset of Haskell embodied by ReWire has been carefully selected to ensure synthesizability in hardware, especially on FPGAs. While a higher order functional language like Haskell has a number of features that are appealing where hardware design is concerned, it contains a number of features that are at best difficult to implement in hardware, and at worst antithetical to efficient hardware design. We identify four main problems (Table 1) where compiling Haskell to hardware is concerned: (1) heap allocation and garbage collection; (2) stack allocation; (3) the existence of undefined (diverging or "crashing") computations; and (4) unpredictable timing behavior. The challenge in designing ReWire is to eliminate these runtime properties by placing suitable restrictions on the semantic features of the language that cause them, while maintaining as much of the expressiveness of Haskell as possible.

2.1 Hardware with Pure Functions and Monads

In this section we explore how to represent hardware circuits in a functional/monadic style. Of necessity arising from space constraints, we assume that the reader has familiarity with Haskell, monads, and monad transformers. Readers requiring more on this topic may wish to refer to the references [21].

Runtime Property	Culprit(s)
Heap allocation/GC	HOF, RDS
Stack allocation	NTR
Divergence/undefinedness	GR, PMF
Unpredictable timing	GR, HOF, RDS

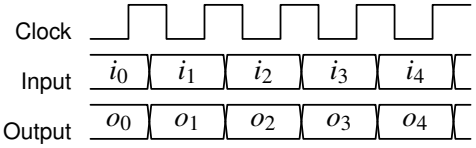
Table 1. Undesirable runtime properties of Haskell, and their semantic antecedents. Key: HOF = higher-order functions; RDS = recursive data structures; NTR = non-tail recursion; GR = general (non-total) recursion; PMF = pattern match failures.

Digital circuit design may be divided roughly into two broad domains: *combinational* circuit design and *sequential* circuit design. Combinational circuits consist only of unclocked logic gates that map one or more binary input signals to one or more binary output signals. Sequential circuits, by contrast, exhibit memory (i.e., the mapping of inputs to outputs changes over time), and are usually tied to a shared clock signal. At a low level, combinational circuits may be implemented purely in terms of logic gates, and sequential circuits may be implemented with a combination of gates and flip flops. In a purely functional language like ReWire, however, we will need higher level abstractions.

Combinational logic represented by pure functions For combinational logic, the choice of representation is straightforward: pure, non-recursive, first-order functions operating on non-recursive first-order data types. Pure functions, i.e., functions which do not have any kind of side effect, are an natural model of combinational circuitry. A binary AND gate, for example, may be expressed as the function `and` in Haskell according to the defining equations:

```
and :: Bit -> Bit -> Bit
and 0 _ = 0
and 1 b = b
```

Sequential logic represented by monadic functions The picture for sequential logic is considerably trickier. To narrow the problem space, we will restrict our attention to circuits with only one clock domain. This enables us to treat the problem somewhat more abstractly, while still covering a very large class of realistic circuit designs. A sequential logic circuit can be viewed as sampling a stream of input values $i_0, i_1, \cdots$ of some type I at each rising (or falling) edge of a clock signal, and producing a stream of output values $o_0, o_1, \cdots$ of some type O in response. The situation is illustrated by the following timing diagram.



For our first attempt at modeling this situation in a functional language, we might consider a simple function $f : I \to O$, but this construction is clearly insufficient to represent sequential circuits with memory, as the response of the circuit to a given input value cannot change over time. As a second attempt, we might consider modeling sequential circuits as functions mapping *lists* of I (i.e., input *histories*) to O, i.e. $f : [I] \to O$. As an abstract mathematical model this does indeed suffice, but it is hard to implement directly (will we need to store the entire input stream history in a RAM?) and does not seem like a very nice structure to program with.

A more realistic possibility is to use a recursive type, something like the Haskell type:

```
data Seql i o = Seql (o,i -> Seql i o)
```

A sequential circuit with inputs of type I and outputs of type O consists of a current output value, and a function that maps an input to a "new" sequential circuit; think of this as a continuation.

This exact structure is, in fact, a monad, but (to make a long story short) the monad instance of this type is not very useful for our particular needs. For ReWire, we will select a slightly different structure, called a *reactive resumption monad*. Reactive resumption monads may be defined in Haskell as follows.

```
newtype React i o a =
  React (Either a (o,i -> React i o a))
return x = React (Left x)
React (Left x)       >>= f = f x
React (Right (o,k)) >>= f =
  React (Right (o,\ i -> k i >>= f))
```

The *React* monad generalizes cleanly to a monad transformer:

```
newtype ReactT i o m a =
  ReactT (m (Either a (o,i -> ReactT i o m a)))
return x = ReactT (return (Left x))
ReactT m >>= f = ReactT (m >>= \ r ->
  case r of
    Left x       -> deReactT (f x)
    Right (o,k) -> return (Right
          ((o, \ i -> k i >>= f)))
 where deReactT (ReactT m) = m
lift m = ReactT (m >>= return . Left)
```

That is, given a base monad m, a computation of type *ReactT i o m a* will compute in m either a result value of type a, or an intermediate output of type o paired with a continuation that is waiting for an input of type i.

One useful convenience function, which we will actually take as a primitive in ReWire, is called `signal`.

```
signal :: Monad m => o -> ReactT i o m i
signal o = ReactT (return (Right (o,return)))
```

This function produces a computation that signals its argument value (type o) on the output, waits until the next input (type i) is available, and returns that value to the caller.

In ReWire, we will use `ReactT` (henceforth abbreviated `ReT`) to express sequential circuits. As with the pure functions we use for combinational circuits, however, a number of restrictions must also be imposed here to ensure compilability. This will be discussed in more detail in Sec. 2.2.

Rounding out the menu of monads, ReWire further contains support for the identity monad (which we will refer to as `I`), as well as a state monad transformer (`StT`); these are equivalent `Identity` and `StateT` in the standard Haskell monad transformer libraries. The specific combination of reactive resumption monads and state monads is provided to enable equational reasoning about information flow, building on previous work that applies these techniques to verifying information flow security [15].

As an aside, we should compare our choice of abstractions to those made by Lava, which is almost certainly the most well-known approach to generating hardware with Haskell. In (at least some versions of) Lava, clock-driven sequential logic is handled as a collection of lazy streams, i.e., infinite demand-driven lists, whose definitions are in effect mutually recursive. If one wishes to program at the level of interacting streams—i.e., to think in terms of interacting signals—this will do the trick. Insofar as the goal of ReWire is to enable monadic equational reasoning, however, the stream-based approach does not suffice. It is not clear, for example, how to leverage the reasoning power offered by layered state monads in the setting of lazy streams. This style of reasoning is essential to our approach to security, as demonstrated in Sec. 4.

2.2 Summary of Language Design

Space limitations preclude a complete, formal description of ReWire's syntax, type system, and semantics here; a full and formal treatment is available in the first author's Ph.D. dissertation [24]. Informally, however, we can define ReWire programs as follows: a ReWire program is a single Haskell module containing (1) zero or more data type declarations, where the data types are *first order* (i.e., they do not have any fields of function type) and *non-recursive*; (2) zero or more *type synonym declarations*; (3) zero or more *"pure" function definitions* whose types are of the form $T_1 \to T_2 \cdots \to T_n \to T$ where $T_1, \cdots, T_n, T$ do not contain function arrows or `ReT`; (4) one or more *reactive function definitions* whose types are of the form

$$T_1 \to T_2 \cdots \to T_n \to \\ \quad ReT\ T_{in}\ T_{out}\ (StT\ T_1^S\ (StT\ T_2^S\ (\cdots (StT\ T_m^S\ I) \cdots)))\ T_{res}$$

where $T_1, \cdots, T_n, T_1^S, \cdots T_m^S, T_{in}, T_{out}, T_{res}$ do not contain function arrows or `ReT`. For a program entry point, a ReWire program must have a reactive function definition named `start` of type $ReT\ T_{in}\ T_{out}\ I\ T_{res}$ for some types T_{in}, T_{out}, and T_{res}.

Recursion is also restricted. "Pure" function definitions as defined above are not allowed to be recursive at all. Reactive function definitions are allowed to be recursive, but they must be tail recursive (i.e., any recursive calls must occur at the very end of a **do**-block), and all recursive calls must be *guarded* [12], which in ReWire means that they must be preceded by a call to **signal**.

A ReWire program is not allowed to import outside packages (including the standard Haskell prelude), but it is always assumed that the following abstract monad operations are available.

$$\begin{array}{lll} \textbf{get} & :: & Monad\ m \Rightarrow StT\ s\ m\ s \\ \textbf{put} & :: & Monad\ m \Rightarrow s \to StT\ s\ m\ () \\ \textbf{signal} & :: & Monad\ m \Rightarrow o \to ReT\ i\ o\ m\ i \\ \textbf{extrude} & :: & Monad\ m \Rightarrow ReT\ i\ o\ (StT\ s\ m)\ a \\ & & \qquad \to s \to ReT\ i\ o\ m\ (a,s) \end{array}$$

The **get** and **put** operations are standard state monad operations. Function **signal** is as defined above. As for **extrude**, this function essentially allows us to supply an initial value to a state-monadic computation; it is akin to `runStT`, but lifts the state monad transformer through `ReT` in the process.

Finally, ReWire programs are allowed to utilize foreign functions written in an external VHDL file via an extended declaration form (somewhat akin to Haskell's foreign function interface). The types of these functions are subject to the same restrictions as "pure" function definitions.

Figure 1 is a complete ReWire program implementing a simple two-function calculator; it will be used as a running example for the discussion of compilation in Sec. 3.

3. Compiling ReWire to VHDL

The ReWire compiler produces circuit implementations from ReWire programs by translating them to state machines implemented in synthesizable VHDL. For the purpose of this discussion, we will divide the process into three phases, illustrated in Fig. 2. The front end (Fig. 2a) is responsible for parsing Haskell concrete syntax and producing a ReWire abstract syntax tree. The current implementation reuses an existing Haskell type checker [18], so further discussion is omitted here. Phases (b) and (c) of Fig. 2 are discussed in Secs. 3.1 and 3.2, respectively.

3.1 Code Generation

Ultimately, the goal of code generation is to produce VHDL code implementing a two-process state machine. Such a machine essentially requires a *single* loop, realized as a VHDL process of the following form:

```
process(clk)
begin
```

```
module Calc where

data Oper = Add W8 | Sub W8 | Clr
type Calc = ReT Oper W8 (StT W8 I)

vhdl plusW8  :: W8 -> W8 -> W8
vhdl minusW8 :: W8 -> W8 -> W8

getVal :: Calc W8
getVal = lift get

putVal :: W8 -> Calc ()
putVal x = lift (put x)

loop :: Calc ()
loop = do x    <- getVal
          oper <- signal x
          case oper of
            Add y -> putVal (plusW8 x y)
            Sub y -> putVal (minusW8 x y)
            Clr   -> putVal 0
          loop

start :: Calc ((),W8)
start = extrude loop 0
```

Figure 1. Running example: a simple two-function calculator.

```
    if clk'event and clk='1' then
      loop body
    end if;
  end process;
```

where the loop body consists entirely of loop-free code. ReWire, however, allows for nested loops (implemented via tail recursion), and the code generation function works simply by emitting code with goto (a construct that does not actually exist in VHDL, though it can be eliminated in a favor of structured programming constructs at a later pass [7]), which means that the single-loop structure we want is not guaranteed to be present. Therefore a "loop flattening" transformation is needed to bring the program into this form. To ease the implementation of this and a few other minor optimizations, the code generation pass (Fig. 2b) generates programs in an imperative intermediate language called PreHDL, whose syntax is defined in Fig. 3. Targeting PreHDL instead of VHDL directly allows us to implement the necessary code transformation passes on a much smaller language than VHDL itself.

An example PreHDL program is given in Fig. 4, corresponding to the compiler's output for the example calculator program of Fig. 1. The (informal) semantics of PreHDL's basic imperative features is standard, with the exception of the **input**, **output**, and **yield** constructs. One may view **input** and **output** as special variables representing respectively the last sampled input value and the current output value of the circuit. The informal meaning of **yield** is to wait until the next clock tick, signaling the current output value on the device output lines in the meantime.

Preliminaries Prior to code generation, the compiler emits a variable declaration for each state monad layer. For example, if `start` is typed in the monad $ReT\ T\ T'\ (StT\ T''\ (StT\ T'''\ I))$, two state variables `s1` and `s0` will be generated, with sizes $sizeof(T'')$ and $sizeof(T''')$, resp. Each recursive reactive function is pre-assigned a code label, as well as argument registers.

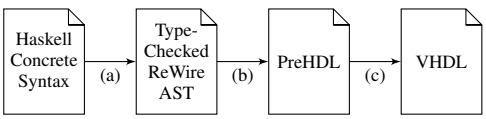


Figure 2. ReWire Compilation Process.

```
Bit      ::= 0 | 1
Int      ::= ( 0 | ··· | 9 )+
Header   ::= IODecl (VarDecl ;)* FunDefn*
IODecl   ::= input : Ty ; output : Ty ;
VarDecl  ::= Name : Ty
Ty       ::= boolean | bits [ Int ]
FunDefn  ::= function Name ( VarDecl , ··· , VarDecl ) {
               (VarDecl ;)* Stmt*
               return Exp ;
             }
           | vhdl Name ( Ty , ··· , Ty ) : Ty;
LHS      ::= Name | input
Stmt     ::= LHS := Exp ; | if BExp { Stmt* } else { Stmt* }
           | label Name : | goto Name ; | yield;
Exp      ::= BExp | Name | Name ( Exp , ··· , Exp )
           | " Bit* " | Exp [ Int : Int ] | concat ( Exp , ··· , Exp )
           | output
BExp     ::= BExp && BExp | BExp || BExp | ! BExp
           | Exp == Exp | Name | true | false
Prog     ::= Header Stmt*
```

Figure 3. PreHDL Syntax.

ReWire's code generator is a function

$$\lceil - \rceil : Env \to ReWireExpr \to Reg \times [PreHDLStmt]$$

where *Env* is a function mapping each ReWire variable currently in scope to the PreHDL register that has been allocated for it. The code generator returns both a sequence of PreHDL statements computing a value, and the register (usually freshly generated) in which the code will store the result. The type *ReWireExpr* represents *typed* expressions, as a few of the compilation cases require type information to resolve overloading of monadic operators. In the actual implementation $\lceil - \rceil$ is defined in terms of a monad, as it requires support for fresh name generation, as well as some pre-computed information about data type sizes and the temporary storage space used for the arguments of reactive functions, but for the presentation here we will just treat these concerns informally. E.g., where a name is required to be fresh we will note this as a side condition.

Example For reference, the PreHDL code that is produced by the compiler for the example of Fig. 1 is presented in Fig. 4. The resulting code has been cleaned up cosmetically for presentation purposes. Note that variable `entrypoint` is introduced during the loop flattening process discussed below; the values it may take are in one-to-one correspondence with the **yield** statements occurring in the unflattened program generated by the code generation phase.

Monadic operators ReWire's monadic operators are simple to compile, as illustrated in Fig. 5. The monadic **return** and **lift** operators are essentially no-ops, and monadic "bind" expressions are compiled exactly as one would compile **let** expressions. State and resumption-monad operators are slightly more complicated. The **get** and **put** operations are compiled in a type-directed fashion, whereby the affected state register is determined by how many state monad transformers are present in the monad transformer stack. A similar scheme applies to **extrude**, which serves to initialize the state monad store variables. The **signal** operator writes its argument value to the output, yields until the next clock tick, and returns the newly sampled input value.

Loop flattening The penultimate compiler pass, following PreHDL generation and preceding VHDL generation, transforms a PreHDL program which may contain **yield** statements at arbitrary program points to a single infinite loop suitable for implementation as a process in VHDL. The basic process is to convert the PreHDL program into a control-flow graph, where **yield** is considered to be

```
input      : bits[10]; output : bits[8];
s0         : bits[8];
entrypoint : bits[1];
vhdl plusW8  (bits[8], bits[8]) : bits[8];
vhdl minusW8 (bits[8], bits[8]) : bits[8];

  entrypoint := "0";
label LOOP:
  if (entrypoint == "0")
    { s0 := "00000000"; }
  else if (entrypoint == "1") {
    if ("00" == input[0:1])
     { s0 := plusW8 (s0,input[2:9]); }
    else if ("01" == input[0:1])
     { s0 := minusW8 (s0,input[2:9]); }
    else
     { s0 := "00000000"; }
  }
  output := s0; ep := "1"; yield; goto LOOP;
```

Figure 4. PreHDL output for the calculator example.

$$
\begin{aligned}
&\lceil \textbf{return}\ e \rceil \rho \quad = \quad \lceil e \rceil \rho \\
&\lceil \textbf{do}\ \{x \leftarrow e\ ;\ e'\} \rceil \rho \\
&\qquad = \quad \langle r_{e'}, K_e\texttt{;}\ K_{e'} \rangle \\
&\qquad \text{where } \langle r_e, K_e \rangle = \lceil e \rceil \rho \\
&\qquad\quad \langle r_{e'}, K_{e'} \rangle = \lceil e' \rceil (\rho[x \mapsto r_e]). \\
&\lceil \textbf{lift}\ e \rceil \rho \quad = \quad \lceil e \rceil \rho \\
&\lceil \textbf{get} : (StT\ T_n\ (\cdots (StT\ T_0\ I)))\ T_n) \rceil \rho \\
&\qquad = \quad \langle r, r\ \texttt{:=}\ sn\texttt{;} \rangle \\
&\qquad \text{where } r \text{ is fresh.} \\
&\lceil \textbf{put}\ e : StT\ T_n\ (\cdots (StT\ T_0\ I))\ ()) \rceil \rho \\
&\qquad = \quad \langle r, K_e\texttt{;}\ \ sn\ \texttt{:=}\ r_e\texttt{;} \rangle \\
&\qquad \text{where } \langle r_e, K_e \rangle = \lceil e \rceil \rho \text{ and } r \text{ is fresh.} \\
&\lceil \textbf{extrude}\ (e : ReT\ T\ T'\ (StT\ T_n\ (\cdots (StT\ T_0\ I)))\ T'')\ e' \rceil \rho \\
&\qquad = \quad \langle r_e, K_{e'}\texttt{;}\ \ sn\ \texttt{:=}\ r_{e'}\texttt{;}\ \ K_e \rangle \\
&\qquad \text{where } \langle r_e, K_e \rangle = \lceil e \rceil \rho \text{ and } \langle r_{e'}, K_{e'} \rangle = \lceil e' \rceil \rho. \\
&\lceil \textbf{signal}\ e \rceil \rho \quad = \quad \langle r,\ K_e\texttt{;} \\
&\qquad\qquad\qquad \texttt{output :=}\, r_e\texttt{;} \\
&\qquad\qquad\qquad \texttt{yield;} \\
&\qquad\qquad\qquad r\ \texttt{:=}\ \texttt{input;} \rangle \\
&\qquad \text{where } \langle r_e, K_e \rangle = \lceil e \rceil \rho \text{ and } r \text{ is fresh.}
\end{aligned}
$$

Figure 5. Compiling monad operations to PreHDL.

a control flow instruction and thus ends a basic block. Given a statement sequence of the form "s_1; `yield`; s_2", s_2 is assigned to a separate basic block from s_1, with a specially marked *yield edge* inserted between them. Due to the guardedness criterion, any well-typed ReWire program will result in a control flow graph in which every cycle contains at least one yield edge. It is therefore possible to *flatten* the control flow graph, i.e., rewrite it into a semantically equivalent graph where there is only one yield edge, and that yield edge is the only back edge in the control flow graph. This flattened control flow graph is translated back into program text. The resulting text may contain goto statements, so we run a final pass to convert those newly introduced goto statements into a structured program [7], resulting in a single infinite loop (possibly preceded by some loop-free initialization code) which contains a single yield statement at the very end. This final form is very easy to translate into a VHDL `process`.

Functions, function calls, and tail calls ReWire functions of "pure"—that is, non-monadic—codomain are simply compiled to functions in PreHDL. For recursive reactive functions, the picture is somewhat more complicated. Each such function is associated with a precomputed label, as well as pre-generated registers that are used to store the argument values (if any). A call to such a function first stores argument values in the function's argument registers, then jumps to its label. This scheme suffices because only tail calls are allowed where reactive-monadic functions are concerned, and, therefore, no stack calling convention is needed.

Pattern matching and data types Since data types are not allowed to be recursive, values of any given data type may be represented by a fixed-width bit vector. Therefore **case** expressions are compiled to bit vector slicing and inspection, and data constructor application to bit vector concatenation. Suppose that data type D has n constructors $C_1 : t_1 \to D, \cdots, C_n : t_n \to D$. (Here we may assume without loss of generality that all data constructors are uncurried.) Then the bit vector used to represent values of type D has two parts. The first part of the vector, the *tag*, carries a distinct value corresponding to data constructor C_i. Tag values are assigned starting from zero in increasing order, so C_1 has tag 0, C_2 has tag 1, and so on. The tag is $\lceil \log_2 n \rceil$ bits wide. The second part of the vector contains the data value supplied with the constructor. For example, if $C_i : Bit \to D$, the data field must be large enough to contain a single bit. If there also exists a constructor $C_j : Bit \times Bit \to D$, the data field needs to be large enough to hold two bits. The overall width of the data field, then, is $max_{0 \le i \le n}(sizeof\ (t_i))$, and the total space needed to store a D value is the sum of the tag width and the data width.

The following table gives a concrete example, corresponding to the `Oper` type of Fig. 1. Here the three constructors require two bits of tag space, and the data field must be eight bits wide to accommodate the `W8` fields for `Add` and `Sub`.

	Tag		Data				
	b_0	b_1	b_2	b_3	$\cdots$	b_8	b_9
`Add x`	0	0	x				
`Sub x`	0	1	x				
`Clr`	1	0	*(don't care)*				
(invalid)	1	1	*(don't care)*				

It is worth noting two things: (1) for constructors that do not use the entire data field, the unused portions of the data field are marked "don't care", meaning that there may be more than one bit vector representation for some values (such as `Clr` in this example), and (2) if the number of data constructors is not a power of 2, there are bit vectors of the proper size that do not represent any valid value of the type. Point (1) is not a problem in practice, as a well-typed ReWire program, by construction, will never inspect the unused bits, and the code we generate will always zero out the unused bits when *producing* a value (thus eliminating the hazard of information leakage when a "long" value is only partially overwritten with a "short" one, and the result copied to the output channel). Point (2) is only an issue where circuit input types are concerned, as the implementation's behavior is undefined if fed an invalid bit string. In practice there are two possible workarounds for this issue: (a) require the user to write a thin wrapper (in ReWire itself or, say, VHDL) that guards against invalid inputs; or (b) ensure that the number of constructors for any input type (as well as the types of all of its data fields, and of its data fields' data fields, and so on) is a power of 2. Option (b) can be implemented as a simple static check; we intend to add a compiler flag to enable this check in the next version of ReWire.

3.2 Generating VHDL from PreHDL

Once a PreHDL program has been converted to the final form exemplified by Fig. 4, translation to VHDL is straightforward. The loop structure of Fig. 4 is replaced with a VHDL `process`, but the loop body is identical except for shallow syntactic differences.

4. Case Study: A Secure Multicore Processor

This section describes the design of a secure dual-core processor in ReWire, complete with a formal statement and proof of a separation policy. The case study illustrates several advantages of the ReWire design paradigm. First, the processor design is *abstract* and *concise*: the monadic design style frees us from the complexity of working with structural hardware primitives. Second, our design is *extensible*: the dual-core version of the processor can be derived from a single-core design in a modular fashion, without any modification or instrumentation of the original design. Third, the design is *formally verified*: the power of equational reasoning, combined with previously published results on the construction and verification of separation kernels in monadic style, allows us to furnish a concise and readable proof of separation. Finally, the ReWire compiler produces an *implementation directly from a high-level specification*: modulo the application of some well-understood semantics-preserving program transformations, the input to the compiler is exactly the artifact we verify in our separation proof; thus there is no semantic gap between the domains of specification, verification, and implementation.

The instruction set architecture of the processor is borrowed from the PicoBlaze 8-bit soft microcontroller from Xilinx [1], specifically its `kcpsm3` iteration that was designed for implementation on Spartan-3 series FPGAs. We selected PicoBlaze primarily to set an ambitious baseline for speed and area comparison: the original PicoBlaze design is constructed in terms of low-level structural primitives that are native to the Spartan-3 architecture (e.g., 2- and 4-input LUTs, and distributed and block RAMs), and its logic was intensively hand-optimized by a highly experienced Xilinx engineer. While it is to be expected that a design implemented in a still-experimental high-level language will fall short of the highly optimized original, we believe that the speed and area tradeoff (discussed in more detail in Sec. 4.1.4) is often acceptable in exchange for the expressive power, extensibility, and ease of formal verification that ReWire provides.

4.1 Single-core Processor

The PicoBlaze ISA is a load/store architecture featuring sixteen 8-bit general purpose registers, a 64-byte scratchpad RAM, and a built-in 32-address control flow stack suitable for procedure calls and interrupt handling [1]. Programs are generally stored in a ROM with a 10-bit address space, and somewhat unusually, instruction words in PicoBlaze are 18 bits wide. I/O is handled via separate 8-bit input and output buses, which are paired with an 8-bit port selection output signal; the actual details of port selection must be handled by a separate circuit outside the PicoBlaze itself.

Figure 6 illustrates the general structure of the ReWire-based PicoBlaze clone. Our design assumes that the register file, the scratchpad RAM, and the control flow stack are implemented as distributed or block RAMs (dual-port in the case of the register file, otherwise single-port) elsewhere on the FPGA. These memories are connected to the ReWire-based module via VHDL port mapping. Input and output signals that cross the outer, dashed block are equivalent to the external interface of the original PicoBlaze. The other signals are connected to the VHDL-instantiated RAMs, which (while not pictured) lie inside the dashed outer block but outside the solid inner block. These RAMs are the only design elements that are implemented outside of ReWire; all instruction processing logic is handled by ReWire. The ReWire-based design presented here diverges somewhat from the original PicoBlaze in terms of instruction cycle timing. In the original implementation, all instructions take precisely two instructions to execute; in the ReWire based design, execution time varies from one to three cycles, with the most commonly used instructions (e.g., arithmetic instructions) taking two cycles.

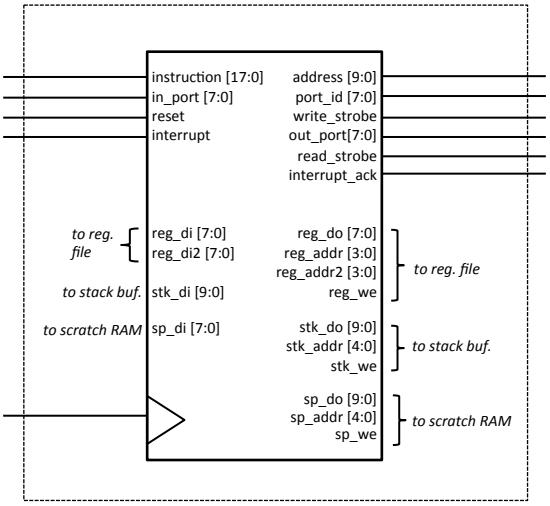


Figure 6. Block diagram of the single core version of the ReWire-based processor. The inner box is the portion implemented in ReWire. Block and distributed RAMs for the register file, stack buffer, and scratchpad RAM are instantiated in VHDL and connected via port mapping.

The remainder of this subsection outlines the design for the single-core processor as written in Haskell. Most of the language features we will be using are also present in ReWire, but the instruction decoder will make light use of function-typed values. For synthesis, this will require a straightforward program transformation called defunctionalization [27], which allows us to convert this program into an equivalent first-order program.

4.1.1 The Monad

The ReWire source code for the processor design, which is available online [25], begins with preliminary type definitions. First, we define a reactive monad for the processor called `CPUM`.

```
type CPUM = ReT Inputs Outputs (StT CPUState I)
```

In other words, the PicoBlaze design lives in a reactive monad with inputs/outputs of type `Inputs` and `Outputs`, and a mutable internal state of type `CPUState`. The `Input` and `Output` types are defined as single-constructor record types, corresponding exactly to the block diagram of Fig. 6 except for the clock signal (which is always implicitly present in ReWire).

```
data Inputs = Inputs { instruction_in :: W18,
 in_port_in    :: W8,   interrupt_in    :: Bit,
 reset_in      :: Bit,  reg_data_in     :: W8,
 reg_data2_in :: W8,    stk_data_in     :: W10,
 sp_data_in    :: W8 }
data Outputs = Outputs {   address_out       :: W10,
 port_id_out        :: W8,  write_strobe_out :: Bit,
 out_port_out       :: W8,  read_strobe_out  :: Bit,
 interrupt_ack_out :: Bit, reg_data_out      :: W8,
 reg_addr_out       :: W4,  reg_addr2_out    :: W4,
 reg_write_out      :: Bit, stk_data_out     :: W10,
 stk_addr_out       :: W5,  stk_write_out    :: Bit,
 sp_data_out        :: W8,  sp_addr_out      :: W6,
 sp_write_out       :: Bit }
```

(Here types of the form Wn refer to n-bit words.) It will sometimes be convenient to have an output record of all zeros, which we will refer to as `out0`.

```
out0 = Outputs { address_out = 0, port_id_out = 0, ... }
```

Finally, the internal state of the processor is a record containing the program counter, the stack pointer, the zero/carry/interrupt-enable flags, and "save" slots for the zero and carry flags (used to temporarily save the flag values when an interrupt occurs).

```
data CPUState = CPUState {
 pc    :: W10, sp    :: W5,
 zFlag :: Bit, cFlag :: Bit, ieFlag :: Bit,
 zSave :: Bit, cSave :: Bit }
```

We omit the definition of various "getter and setter" methods for the individual state fields, as well as convenience functions `incrPC` and `incrSP` to increment the program counter and stack pointer.

4.1.2 Instruction Decoding

The instruction decoder takes the form of a pure function `decode` from instruction words of type `W18` to an algebraic data type `Instr`, which provides a semantically structured representation of each instruction's action:

```
data Instr =
   Binop Binop Reg Rand | Branch Bit Cond W10
 | Return Cond          | Returni Bit
 | IEnable Bit          | Fetch Reg Rand
 | Store Reg Rand       | Input Reg Rand
 | Output Reg Rand      | Invalid
type Binop = W8 -> W8 -> Bit -> Bit -> (W8,Bit,Bit)
data Rand  = ConstRand W8 | RegRand Reg
data Cond  = NoCond | CCond | NCCond | ZCond | NZCond
type Reg   = W4
```

The constructors respectively represent arithmetic/logical instructions such as `ADD`; branch instructions (possibly conditional); return instructions (again, possibly conditional); the return-from-interrupt instruction; the interrupt-enable instruction; fetch/store instructions; input/output instructions; and a catch-all case for any invalid instruction words. Note that the type `Binop`, which represents the particular operation being requested, has function type; specifically, an arithmetic/logical operation is represented by a function taking two `W8`-typed operands and the initial value of the `Z` and `C` flags as arguments, and returning the `W8`-typed result along with the new value for `Z` and `C`. Since the definition of `decode` consists entirely of routine pattern matching on bit vectors and construction of `Instr`s, we shall omit it here.

4.1.3 Main Loop and Startup

The processor's execution is structured as a loop. The argument of type `Inputs` threads through the last received input value.

```
loop :: Inputs -> CPUM ()
```

The basic form of the loop is as follows:

```
loop i = case reset_in i of
  1 -> -- ...handle reset...
  0 -> do
   ie <- getIEFlag
   case (ie,interrupt_in i) of
    (1,1) -> -- ...handle interrupt...
    _     -> case decode (instruction_in i) of
     BinopInstr ...   -> -- ...handle arith instr...  (*)
     BranchInstr ...  -> -- ...handle branch instr...
        ...
     OutputInstr ...  -> -- ...handle output instr...
     InvalidInstr ... -> -- ...handle invalid instr...
```

where each of the elided codepaths ultimately results in a guarded tail call back to `loop`.

For space reasons, we shall examine only the case of arithmetic/logical instructions (i.e., the line marked `(*)` in the above code listing). In the first clock cycle, we increment the value of the program counter, and signal the needed register indices on the register file address lines. Note that we assume a dual-port RAM for the register file; we may therefore fetch both the registers `rx` and `ry` in one clock cycle.

```
BinopInstr o rx rand -> do
  incrPC
  i <- signal (out0 { reg_addr_out  = rx,
                      reg_addr2_out = case rand of
                        ConstRand _ -> 0
                        RegRand ry  -> ry })
```

In the second cycle, we must first compute the result values from the operation. After fetching the current value of the zero and carry flags (`zf` and `cf`) and the operand values (`vx` and `vy`) from the input lines (or from the instruction word if the `vy` is a constant), we feed these values to the function `o`, producing a result value `r` and new values `zf'` and `cf'` for the zero and carry flags.

```
  zf <- getZFlag
  cf <- getCFlag
  let vx          = reg_data_in i
      vy          = case rand of
                      ConstRand k -> k
                      _           -> reg_data2_in i
      (r,zf',cf') = o vx vy zf cf
```

We then write back the new values of the `Z` and `C` flags.

```
  putZFlag zf'
  putCFlag cf'
```

Finally, we signal for the next instruction, simultaneously writing the new value for register `rx` back to the register file, and tail-recursively return to the top of the `loop`.

```
  pc <- getPC
  i  <- signal (out0 { address_out   = pc,
                       reg_addr_out  = rx,
                       reg_write_out = 1,
                       reg_data_out  = r })
  loop i
```

Now with the `loop` defined, the top-level entry point is `start`, which signals an "empty" output, and enters the `loop`.

```
start :: ReT Inputs Outputs I ((),CPUState)
start = do i <- signal out0
           extrude (loop i)
```

4.1.4 Synthesis

Defunctionalization As mentioned above, the Haskell version of the processor specification contains a handful of higher-order functional constructs. This is not allowed in ReWire, so we must transform the program into a first-order form before we can synthesize a circuit. As it happens, there exists a program transformation due to Reynolds [27] called *defunctionalization* that allows us to do just this in a straightforward, mechanical way. In this example, the upshot of defunctionalization is that all functions of type `Binop` will be replaced with values in a data type that represents all `Binop` functions used in the program, and any calls to such functions are replaced with calls to an interpretation function. This suffices to produce a program that is compilable by ReWire.

Synthesis results To evaluate performance, both the ReWire-based processor described here and the original PicoBlaze from Xilinx were synthesized using the XST synthesis tool for a Xilinx Spartan-3E XC3S500E, speed grade -4. XST was configured to optimize for speed (as opposed to area), with normal optimization effort. Synthesis estimates for device utilization and F_{max} follow.

	Slices	Flip Flops	4-LUTs	F_{max} (MHz)
PicoBlaze	99	76	181	139.919
ReWire	451	110	866	69.956

Put another way, the ReWire-based processor is approximately 4.6 times as large as the original (as measured in slices), and is capable of operating at about half the maximum clock speed. We believe these performance results, within an order of magnitude of

the original, are quite promising for two reasons. First, PicoBlaze is a very low-level design that was heavily optimized by an experienced engineer employed by Xilinx. Thus it is to be expected that any design of a high-level behavioral flavor will fall short of the original on performance. Second, the ReWire compiler is still in a very early development stage and does virtually no optimization of the resulting VHDL before handing it off to XST. As work proceeds on more aggressive optimization, we expect that the complexity of the combinational logic emitted by ReWire will be reduced substantially. This should bring the size and performance overhead into a range that will be quite acceptable for many users in exchange for the high assurance capabilities of ReWire.

4.2 Secure Multi-domain Processor

We now demonstrate how the single-core processor of Sec. 4.1 may be converted into a dual-core processor with secure shared state. In particular, we will extend the design of Sec. 4.1 essentially by instantiating two copies of the processor core, and wrapping them with a secure harness whose design is akin to a software-based monadic separation kernel [15]. One of these cores will be designated as the "high" core and the other as the "low" core, reflecting different security levels in a lattice. We also insert a shared 8-bit register mapped to I/O port 0xFF, which the low core may write to and the high core may read (but not write). Any attempt by the individual cores to access port 0xFF will be mediated by the harness, which will ignore write requests from the high core.

4.2.1 The Dual-Core Harness

The dual-core harness serves to "lift" the individual cores into a *layered state monad* [15]. As will become clear in Sec. 4.3, the application of multiple state monad transformers provides a useful basis for reasoning about the separation of state domains. For the dual-core harness we will provide three layers of state: one (of type `W8`) for the shared register, one for the high core's internal state, and one for the low core's internal state. We will also provide separate input and output channels for the high and low cores, meaning that the input and output types become pairs. Thus we arrive at the "dual-core monad" (`DCM`), where the harness lives:

```
type DCM = ReT (Inputs,Inputs) (Outputs,Outputs) K
type K   = StT W8 (StT CPUState (StT CPUState I))
```

The `harness` will operate in a tail recursive fashion, taking two `CPUM` computations reflecting the execution current state of the high and low cores respectively, and producing a computation in `DCM`. (Note that the definition of `harness` utilizes a number of helper functions that will be explained below.) The harness proceeds by running each core for a single step against its respective state layer. If either of the cores has halted execution (which never actually happens with the cores we are considering), we halt the overall system as well. Otherwise, the harness forwards the output signals of the individual cores to the outside world via a **signal** call. When the next input signal is obtained, the helper functions `checkHiPort` and `checkLoPort` serve to filter requests for the shared register; if the low core attempts to write, the request value will be written to the shared register, and if the high core attempts to read the shared register, the value on its input port will be overwritten with the value of the shared register. The harness then feeds the filtered inputs to the cores and returns (via tail recursion) to the top of the loop.

```
harness :: CPUM a -> CPUM b -> DCM (Either a b)
harness lo hi = do
  r_lo <- lift (liftKL (deReT lo))
  r_hi <- lift (liftKH (deReT hi))
  case (r_lo,r_hi) of
    (Left a,_) -> return (Left a)
    (_,Left b) -> return (Right b)
    (Right (o,k_lo),Right (o_hi,k_hi)) -> do
      (i_lo,i_hi) <- signal (o_lo,o_hi)
      i_hi'       <- checkHiPort i_hi o_hi
      checkLoPort o_lo
      harness (k_lo i_lo) (k_hi i_hi')
```

The helper functions `liftKL` and `liftKH` allow state actions of the individual cores to be mapped onto a single state domain in the layered monad. They are defined as follows.

```
liftKL :: StT CPUState I a -> K a
liftKL m = lift (lift m)

liftKH :: StT CPUState I a -> K a
liftKH m = lift (do
  s <- get
  let (a,s') = runI (runStT m s)
  put s'
  return a)
```

For `checkHiPort`, we pattern match on the output value of the high core; if it contains a read request for address 0xFF, we pull the value out of the shared register and overwrite the data input for the high core with that value. Otherwise the input is left unmodified.

```
checkHiPort :: Inputs -> Outputs -> DCM Inputs
checkHiPort = lift (
  case (port_id_out o_hi,read_strobe_out o_hi) of
    (0xFF,1) -> do
      v <- get
      return (i_hi { in_port_in = v })
    _        -> return i_hi)
```

Dually, `checkLoPort` translates write requests from the low core into a write on the shared register.

```
checkLoPort :: Outputs -> DCM ()
checkLoPort o_lo = lift (
  case (port_id_out o_lo,write_strobe_out o_lo) of
    (0xFF,1) -> put (out_port_out o_lo)
    _        -> return ())
```

This completes the design of the secure dual-core processor.

4.2.2 Synthesis

The dual-core processor makes much more extensive use of higher-order language features than the single-core. In particular, the harness loop takes two monadic computations as arguments. Nevertheless, defunctionalization still suffices to transform this specification into a first-order, compilable form. The fully defunctionalized version of the processor and harness are available in the code repository [25].

Synthesis results Synthesis estimates for device utilization and maximum clock speed of the dual-core processor were obtained by the same process used for the single-core processor in Sec. 4.1.4. The following table illustrates the results; slice utilization, flip flop utilization, LUT utilization, and F_{max} are given for the dual-core processor in the first row, with the prior results for the single-core processor given in the second row. The third row reports the ratio for each metric between the dual- and single-core processors.

	Slices	Flip Flops	4-LUTs	F_{max} (MHz)
2-Core	907	258	1735	67.867
1-Core	451	110	866	69.956
Ratio	2.011	2.345	2.003	0.970

The results suggest nearly ideal scaling. Slice and LUT utilization for the dual core processor are almost exactly twice as much as the single core processor, while flip flop utilization suffers a slight penalty attributable to the extra state registers required for the harness to track its own internal state. The timing burden imposed by dual core support is also minimal: maximum frequency of the dual-core processor is within 3% of the single-core processor.

4.3 Proof of Separation

To specify the security of the harness, we apply a security model developed for modular monadic semantics called *take separation* [15]. With this approach, the operation of the (`harness lo hi`) is compared to the operation of (`harness lo skip`), where `skip` is a "nop" core. The basic standard of security requires that both systems, when executed on the same finite input traces, should produce identical `lo` outputs. The following defines the "nop" core:

```
skip :: Outputs -> Inputs -> ReT Inputs Outputs K a
skip o i = ReT (return (Right (o,skip o)))
```

N.b., that the core (`skip o i`) produces a constant output and entirely ignores its input.

The `pull` function runs a system on a finite list of dual inputs:

```
pull :: [Outputs] -> [(Inputs,Inputs)] ->
  ReT (Inputs,Inputs) (Outputs,Outputs) K [Outputs] ->
  K [Outputs]
pull os [] _       = return os
pull os (i:is) phi = next phi >>= \ (Right (o,k)) ->
                     pull (os ++ [fst o]) is (k i)
  where next = deReT
```

The function call, (`pull os is` (`harness lo hi`)), executes the two core system on input `is` and accumulates each `lo` output in order on the first argument. N.b., that we are assuming, without loss of generality, that the system (i.e., `pull`'s third argument) never terminates (i.e., always returns a `Right`). When the input list is exhausted, the accumulated `lo` outputs are returned.

Theorem 1 states the security specification of the harness system. In it, the operation of (`harness lo hi`) is compared to that of (`harness lo` (`skip` $\mathtt{o}_0$ $\mathtt{i}_0$)) within the context of "$\ldots$ >>= κ_0". The purpose of the initial continuation κ_0 is to screen out any of `hi`'s effects on both sides of the equation while still returning `lo`'s outputs. This is analogous to the role of projecting out high level operations in a conventional, event based security model [13]. The full proof of Theorem 1 is available in an online supplement to this paper [25].

Theorem 1 (Harness Security)**.** *For any appropriately typed* $\mathtt{i}_0$, $\mathtt{o}_0$*, os, finite is, lo and hi,*

$$\begin{aligned}&\mathtt{pull\ os\ is}\ (\mathtt{harness\ lo\ hi}) \mathbin{>\!\!>\!=} \kappa_0\\ &\quad = \mathtt{pull\ os\ is}\ (\mathtt{harness\ lo}\ (\mathtt{skip}\ \mathtt{o}_0\ \mathtt{i}_0)) \mathbin{>\!\!>\!=} \kappa_0\\ &\textbf{where}\\ &\quad \kappa_0 = \lambda \mathtt{os}.\ \mathit{mask}_{\mathtt{H}} \mathbin{>\!\!>} \mathtt{return\ os}\end{aligned}$$

□

5. Conclusions and Future Work

This paper has presented ReWire, a functional programming language and compiler for synthesizing efficient hardware circuits from modular, high-level, semantics-directed designs. ReWire is both a computational λ-calculus suitable for writing formal specifications and an expressive functional language and compiler for generating efficient hardware artifacts. The hypothesis of this work is that this duality will position ReWire to avoid the pitfalls of semantic archaeology without sacrificing performance. With ReWire, the text of a design is verified (rather than a reconstructed model of the design) and the compiler transforms that same design into hardware, thereby unifying the languages of specification, design and implementation. As the case study of Sec. 4 demonstrates, this design paradigm brings great benefits with respect to the modular construction and formal verification of hardware.

Related Work. Delite is a compiler framework and runtime for parallel embedded domain-specific languages (EDSLs) that has been retargeted to produce hardware [9]. Like Delite, ReWire is a DSL embedded in a functional language—in our case, Haskell; in theirs, Scala. Both ReWire and Delite are virtualized DSLs, meaning that they reuse substantial portions from their respective host language's front ends. As virtualized DSLs, both Delite and ReWire exhibit what Delite's creators call "the three P's" [19]: productivity, performance and portability. But ReWire is based in modular monadic semantics [21] applied to concurrency [16], and so it exhibits a fourth "P": provability. Previous work [15, 17] demonstrates the utility of monadic types and structures to verifying security and safety properties.

Edwards [5] has commented on the difficulty of compiling from a C like language to hardware. This has led him to pursue Haskell as a source [6]. The use of functional abstractions, such as monads, greatly speeds the construction of complex circuits, and makes their specifications more extensible. Lazy pure functional languages readily accommodate parallelism; e.g., in $(\mathtt{e}_1, \mathtt{e}_2)$, the subexpressions $\mathtt{e}_1$ and $\mathtt{e}_2$ may be safely evaluated in parallel due to the absence of side effects. Reactive resumption monads as used in ReWire refine this inherent parallelism with a notion of interactivity and a notion of lock-step or clocked sequentiality. C, possessing no inherent notion of timing granularity, does not lend itself to the notions of computation found in hardware.

Other projects have explored the use of Haskell as a source language for hardware synthesis. CλaSH [2] is a compiler for a subset of Haskell to VHDL. Like ReWire, CλaSH uses Haskell itself as a source language and requires some limits be placed on the kinds of algebraic data types used as well as the basic operating types. ForSyDe is a platform to compile models of hardware written in Haskell to circuitry [29]. Neither CλaSH nor ForSyDe, however, makes use of the modular monadic abstractions that are essential to ReWire's verification approach.

Hardware description languages (HDL) have been, and continue to be, an area of active research. Lava is a family of domain-specific languages for hardware specification embedded in Haskell [3, 10]. Primitives in Lava are structural and specify circuits at the level of signals. ReWire, by contrast, compiles a subset of Haskell itself to hardware circuits, including control flow constructs that are difficult to capture in deeply embedded DSLs [11], and relies on an abstract set of behavioral primitives.

Caisson [20] is based on a classic security type system [32] for a Verilog-like hardware definition language. Therefore, security in Caisson inherits the strengths and weaknesses of the type-based approach; i.e., security is statically checkable, but some secure programs are excluded. Caisson programs can be transliterated into Verilog and thereby synthesized to hardware. ReWire's design adheres to the DSL philosophy: its design is flexible and agile. ReWire is a functional monadic language and inherits the advantages of that style from Haskell: modularity and extensibility of the language itself. Specifications structured with monads come with "by-construction" properties useful for formal verification [15].

Recent research has demonstrated the value of monadic semantics to the formal specification and verification of x86- and ARM-based systems [8, 30]. These efforts are not, however, focused on producing synthesizable artifacts; monads are used purely as a vehicle for reasoning.

Future Work. One benefit of the functional-language approach to hardware is that functional languages are generally amenable to formal specification. However, a drawback of the approach is that hardware engineers are not typically well-versed in functional languages (there are exceptions, of course). While this drawback is social in nature, it is still significant. Hardware engineers frequently view designs in graphical terms. We are planning to build a graphical front end to ReWire to aid hardware engineers and encourage adoption of the ReWire tools.

Other future research directions we are pursuing have to do with increasing the expressiveness of the type system to support

metaprogramming, as well as type-based enforcement of information flow policies. There are type systems for staged programming (e.g., MetaML [31]) that we believe will improve programmer productivity further while maintaining type safety. Staging annotations enable programmers to safely encode source-level transformations and optimizations. Previous work has focused on type systems for enforcing fault isolation in calculi based on reactive resumptions [17]; we believe that a similar strategy may be employed to enforce information flow security.

Another avenue of future work is to adapt the ReWire compiler to enable programs that mix CPU and FPGA-based computation. A sizable portion of the Haskell language—basically anything involving recursion at runtime—is not synthesizable with ReWire. A mixed-mode compiler could take the non-synthesizable portions of the program and compile them for use on a CPU-based system containing an FPGA, with the two parts of the program communicating over the system bus. We anticipate that reactive resumptions would provide a powerful tool for tackling the coordination challenges inherent to such heterogeneous systems.

6. Acknowledgements

This research has been supported by the Office of the Assistant Secretary of Defense for Research and Engineering, by the U.S. Department of Education under GAANN grant number P200A100053, and by NSF CAREER Award 00017806.

References

[1] *PicoBlaze 8-bit Embedded Microcontroller User Guide*. Xilinx, Inc., 2011.

[2] C. Baaij and J. Kuper. Using rewriting to synthesize functional languages to digital circuits. In *Trends in Fun. Prog.*, volume 8322 of *LNCS*, pages 17–33, 2014.

[3] P. Bjesse, K. Claessen, and M. Sheeran. Lava: Hardware design in Haskell. In *ICFP '98*, pages 174–184, 1998.

[4] D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 167–182, 2008.

[5] S. A. Edwards. The challenges of synthesizing hardware from C-like languages. *IEEE Design and Test of Computers*, 23(5):375–386, 2006.

[6] S. A. Edwards. A finer functional Fibonacci on a fast FPGA. Technical Report CUCS-005-13, Department of Computer Science, Columbia University, February 2013.

[7] A. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *In Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994.

[8] A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the First International Conference on Interactive Theorem Proving*, ITP'10, pages 243–258, 2010.

[9] N. George, H. Lee, D. Novo, T. Rompf, K. Brown, A. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Proc. of 24th Int. Conf. on Field Prog. Logic and App. (FPL '14)*.

[10] A. Gill. Declarative FPGA circuit synthesis using Kansas Lava. In *ERSA '11*, 2011.

[11] A. Gill. Domain-specific languages and code synthesis using Haskell. *ACM Queue*, 12(4):30:30–30:43, Apr. 2014.

[12] C. E. Giménez. *Un Calcul De Constructions Infinies Et Son Application A La Verification De Systemes Communicants*. PhD thesis, L'École Normale Supérieure de Lyon, 1996.

[13] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. of the 1982 Symposium on Security and Privacy (SSP '82)*, pages 11–20. IEEE Computer Society Press, 1990.

[14] S. Goncharov and L. Schröder. A coinductive calculus for asynchronous side-effecting processes. In *Proceedings of the 18th International Conference on Fundamentals of Computation Theory*, pages 276–287, 2011.

[15] W. L. Harrison and J. Hook. Achieving information flow security through monadic control of effects. *Journal of Computer Security*, 17(5):599–653, 2009.

[16] W. L. Harrison and A. Procter. Cheap (but functional) threads. 44 pages. Accepted for publication in *Higher-Order and Symbolic Computation*.

[17] W. L. Harrison, A. Procter, and G. Allwein. The confinement problem in the presence of faults. In *Proceedings of the 14th International Conference on Formal Engineering Methods*, ICFEM'12, pages 182–197, 2012.

[18] M. P. Jones. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, pages 68–78, Paris, France, 21–24 Oct. 1999.

[19] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, Sept. 2011. ISSN 0272-1732.

[20] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: a hardware description language for secure information flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM.

[21] S. Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1998.

[22] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[23] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[24] A. Procter. *Semantics-Driven Design and Implementation of High-Assurance Hardware*. PhD thesis, University of Missouri, 2014.

[25] A. Procter, W. L. Harrison, I. Graves, M. Becchi, and G. Allwein. Online supplement accompanying "Semantics-driven hardware design, implementation, and verification with ReWire". URL http://adamprocter.com/lctes15.

[26] A. Procter, W. L. Harrison, I. Graves, M. Becchi, and G. Allwein. Semantics-directed machine architecture in ReWire. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT'13)*, pages 446–449, December 2013.

[27] J. Reynolds. Definitional interpreters for higher order programming languages. *ACM Conference Proceedings*, pages 717–740, 1972.

[28] J. Rushby. Design and verification of secure systems. In *Proceedings of the ACM Symposium on Operating System Principles*, volume 15, pages 12–21, 1981.

[29] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, 2004.

[30] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 379–391, 2009.

[31] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(12):211–242, 2000.

[32] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2–3):167–187, 1996.

[33] M. Wilding, D. Greve, R. Richards, and D. Hardin. Formal verification of partition management for the AAMP7G microprocessor. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175–191. 2010.