A Principled Approach to Secure Multi-Core Processor Design with ReWire

ADAM PROCTER, University of Missouri
WILLIAM L. HARRISON, University of Missouri
IAN GRAVES, University of Missouri
MICHELA BECCHI, University of Missouri
GERARD ALLWEIN, U.S. Naval Research Laboratory

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 advocates a *semantics-directed* approach to bridge this conceptual gap. We present a case study in the design of secure processors, which are formally derived via principled techniques grounded in functional programming and equational reasoning. The case study comprises the development of secure single- and dual-core variants of a single processor, both based on a common semantic specification of the ISA. We demonstrate via formal equational reasoning that the dual-core processor respects a "no-write-down" information flow policy. The semantics-directed approach enables a modular and extensible style of system design and verification. The secure processors require only a very small amount of additional code to specify and implement, and their security verification arguments are concise and readable. Our approach rests critically on ReWire, a functional programming language providing a suitable foundation for formal verification of hardware designs. This case study demonstrates both ReWire's expressiveness as a programming language and its power as a framework for formal, high-level reasoning about hardware systems.

CCS Concepts: •**Security and privacy** → **Logic and verification; Security in hardware;** •**Hardware** → **Hardware description languages and compilation; Functional verification;** •**Software and its engineering** → **Functional languages;**

Additional Key Words and Phrases: Equational reasoning, monads, hardware security, reconfigurable computing

ACM Reference Format:
Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein, 2015. A Principled Approach to Secure Multi-Core Processor Design with ReWire. *ACM Trans. Embedd. Comput. Syst.* V, N, Article A (January YYYY), 25 pages.
DOI: 0000001.0000001

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.
Author's addresses: A. Procter, Computer Science Department, University of Missouri; W. L. Harrison, Computer Science Department, University of Missouri; I. Graves, Computer Science Department, University of Missouri; M. Becchi, Electrical and Computer Engineering Department, University of Missouri; G. Allwein, U.S. Naval Research Laboratory, Washington, D.C.

DOI: 0000001.0000001

1. INTRODUCTION

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 built 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). The ReWire functional language [Procter et al. 2015b] eliminates the need for semantic archaeology by enabling a "semantics-first" design style, providing a source language with clear semantic foundations, a formal framework for reasoning about security, and a compiler that produces efficient implementations.

In this work, we demonstrate the ReWire methodology, which is based on modular monadic semantics [Liang et al. 1995; Harrison et al. 2009], via an in-depth case study in the implementation and verification of secure processors. First, we develop a semantic specification of an instruction set, which is synthesizable directly to a single-core implementation of that ISA. Second, we use the same specification to derive a *secure* single-core variant of this processor, with a shared execution unit that does not introduce information flow between security domains. Third and finally, we develop from the same specification a secure *dual-core* version of the processor, which implements an internal storage channel for inter-core communication, and verify via equational reasoning that the dual-core processor respects a "no-write-down" security policy.

The benefits of implementing separation in hardware are substantial [Popek and Goldberg 1974; Wilding et al. 2010], because hardware-based security mechanisms provide greater efficiency and increased confidence in system properties. The technical burden of constructing and verifying a separating processor is, however, quite substantial. The case study in the present work demonstrates that ReWire substantially lightens that burden by avoiding many of the pitfalls of traditional formal methods practice. In particular, ReWire eliminates the need for a separate formal model of the system that is independent of its implementation and, therefore, the possibility that the formal system model does not actually reflect the behavior of the implementation.

The key contributions of this article 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) 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. This work extends a previous conference paper by the authors [Procter et al. 2015b] with a more in-depth treatment of the case study, including a formal security proof not previously published. More details on the implementation of the ReWire compiler may be found in that paper.

The remainder of the article proceeds as follows. Section 2 discusses related work. An overview of the design of the ReWire language is given in Section 3. Section 4 outlines a case study in the design of secure processors in ReWire. Section 5 details the verification of these processors, and Section 6 discusses the performance characteristics of circuits generated by the ReWire compiler. We conclude with a discussion of future work in Section 7.

2. RELATED WORK

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 [Harrison and Hook 2009; Harrison et al. 2012; Harrison and Procter 2015]. As design tools, these ideas have many virtues. They are flexible and expressive, support formal analysis, and can be readily evaluated with any Haskell implementation. In this article we demonstrate that modular monadic semantics, previously applied to software artifacts such as interpreters and compilers [Liang 1998] and operating system kernels [Cock et al. 2008; Harrison and Hook 2009], is also useful in the realm of hardware design.

ReWire is a formally defined programming language for expressing reactive, concurrent, and parallel computations. ReWire is a computational λ-calculus [Moggi 1991] and, as such, is conducive to formal verification [Goncharov and Schröder 2011] of, in particular, security and safety properties [Harrison and Hook 2009; Harrison et al. 2012]. 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 present a substantial case study demonstrating that ReWire supports the rapid development and implementation of provably secure hardware.

Delite is a compiler framework and runtime for parallel embedded domain-specific languages (EDSLs) that has been retargeted to produce hardware [George et al. 2014]. 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 of their respective host language's front ends. As virtualized DSLs, both Delite and ReWire exhibit what Delite's creators call "the three P's" [Lee et al. 2011]: productivity, performance and portability. But ReWire is based in modular monadic semantics [Liang 1998] applied to concurrency [Harrison and Procter 2015], and so it exhibits a fourth "P": provability. Previous work [Harrison and Hook 2009; Harrison et al. 2012] demonstrates the utility of monadic types and structures to verifying security and safety properties.

Edwards has commented on the difficulty of compiling from a C like language to hardware [Edwards 2006]. This has led him and other collaborators to pursue Haskell as a source [Zhai et al. 2015]. 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 [Baaij and Kuper 2014] 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. Chisel [Bachrach et al. 2012] takes a similar approach, but is embedded in Scala rather than Haskell. ForSyDe is a platform to compile models of hardware written in Haskell to circuitry [Sander and Jantsch 2004]. Neither CλaSH nor ForSyDe, however, enables 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 [Bjesse et al. 1998; Gill 2011]. Primitives in Lava specify circuits

at the level of signals, with Haskell essentially playing the role of a metaprogramming language. 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 [Gill 2014], and relies on an abstract set of behavioral primitives.

Caisson [Li et al. 2011] is a variant of Verilog that implements on a classic security type system [Volpano et al. 1996]. Therefore, security in Caisson inherits the strengths and weaknesses of the purely 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, by contrast, is a functional monadic language and inherits the advantages of that style from Haskell: modularity and extensibility of the language itself. ReWire's design adheres to the DSL philosophy: its design is flexible and agile. While ReWire also embraces strong static typing, specifications structured with monads also come with "by-construction" properties useful for formal verification [Harrison and Hook 2009].

Recent research has demonstrated the value of monadic semantics to the formal specification and verification of x86- and ARM-based systems [Sarkar et al. 2009; Fox and Myreen 2010]. In contrast to the present work, however, these efforts are not focused on producing synthesizable artifacts; monads are used purely as a vehicle for reasoning.

3. OVERVIEW 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 stages of design 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. Due to space constraints, the remainder of this section summarizes ReWire's design only at a high level. Further details may be found in previous publications [Procter 2014; Procter et al. 2015b].

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, many other of its features are at best difficult to implement in hardware, and at worst antithetical to efficient hardware design. We identify four main problems (Figure 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.

3.1. 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 famil-

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

Fig. 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.

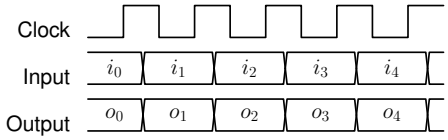
iarity with Haskell, monads, and monad transformers. Readers requiring more background on this topic may wish to refer to the references [Liang 1998].

Digital circuit design may be divided 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 a 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. Assuming a single clock domain, 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 using 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 something like the recursive type:

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

In other words, a sequential computation 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 computation; think of this function as a continuation.

This exact structure is, in fact, a monad, but (to make a long story short) it is not a very useful monad 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 Re i o a       = Re (Either a (o,i -> Re i o a))
Re (Left x)      >>= f = f x
Re (Right (o,k)) >>= f = Re (Right (o,\ i -> k i >>= f))
return x               = Re (Left x)
```

The `Re` monad may be generalized as a monad transformer:

```
newtype ReT i o m a = ReT (m (Either a (o,i -> ReT i o m a)))

return x      = ReT (return (Left x))
ReT m >>= f   = ReT (m >>= \ r -> case r of
                      Left x      -> deReT (f x)
                      Right (o,k) -> return (Right ((o, \ i -> k i >>= f)))
deReT (ReT m) = m
lift m        = ReT (m >>= return . Left)
```

That is, given a base monad `m`, a computation of type `ReT 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 -> ReT i o m i
signal o = ReT (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.

As with the pure functions we use for combinational circuits, a number of restrictions must be imposed here to ensure compilability. These restrictions will be discussed in more detail in Sec. 3.2. 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 to `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 [Harrison and Hook 2009].

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. 5.

3.2. 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 [Procter 2014]. 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 $T_1 \rightarrow T_2 \cdots \rightarrow T_n \rightarrow T$ where $T_1, \cdots, T_n, T$ do not contain function arrows, `StT`, or `ReT`; (4) one or more *reactive function definitions* whose types are of the form

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

where $T_1, \cdots, T_n, T_1^S, \cdots T_m^S, T_{in}, T_{out}, T_{res}$ do not contain function arrows, `StT`, 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 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* [Giménez 1996], which in ReWire means that they must be syntactically 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 abstract monad operations of Fig. 2 (bottom) are present. The **get** and **put** operations are standard state monad operations. Function **signal** has Haskell semantics as defined above. As for **extrude**, this function essentially allows us to supply an initial value to a resumption-and-state-monadic computation; it is akin to Haskell's `runStateT`, 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.

The example program of Figure 2 (top) is a complete ReWire program (a simplified two-function calculator) that demonstrates usage of all of the above features. The program lives in a state-and-resumption monad called `Calc`, which stores the current value on the calculator's "display" in a state variable, takes an input of type `Oper` at each clock tick, and produces an output of type `W8` (8-bit word) at each cycle. An `Oper` is either an addition or subtraction operation (`Add` or `Sub`, both of which carry `W8`-typed operands), or a clear operation (`Clr`). The addition and subtraction functions `plusW8` and `minusW8` are presumed to be defined externally in VHDL. The basic structure of the program is a tail recursive `loop` which repeatedly: retrieves the current stored value from the state monad via a call to `getVal`; signals the current stored value on the output with a call to `signal` and retrieves a new input at the next clock tick; mutates the state based on the "opcode" of the new input; and returns to the top of the `loop`.

4. DERIVING SECURE PROCESSOR DESIGNS

This section describes the design of two secure processors in ReWire. The case study illustrates several advantages of the ReWire design paradigm. First, the processor designs are *abstract*, *concise*, and *extensible*. The monadic design style frees us from the complexity of working with structural hardware primitives, and the secure processor specifications can be derived from a single-core reference design in a modular fash-

```
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
```

```
get     :: Monad m ⇒ StT s m s
put     :: Monad m ⇒ s → StT s m ()
signal  :: Monad m ⇒ o → ReT i o m i
extrude :: Monad m ⇒ ReT i o (StT s m) a → s → ReT i o m (a, s)
```

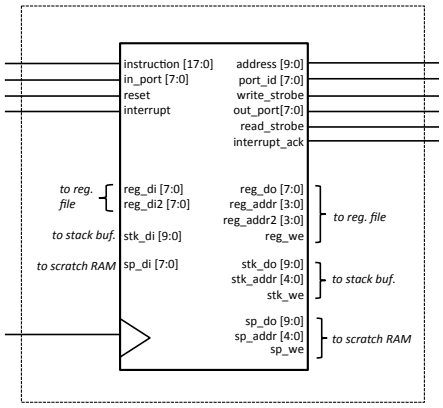
Fig. 2. (top) Example ReWire program: a simple two-function calculator. (bottom): Type signatures of monadic primitives built-in to ReWire.

ion, without any modification or instrumentation of the original design. Second, as we will see in Sec. 5, the dual-core design is *formally verified*: the power of equational reasoning 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. Sec. 6 demonstrates that this high-level approach comes at a reasonable cost with respect to speed and circuit size.

At the design stage, we begin with a high-level semantic specification of the instruction set architecture written in ReWire. This specification serves a dual role as an *executable* instruction set interpreter (via Haskell), as well as a *synthesizable* circuit design (via ReWire). Our ISA specification is outlined in Sec. 4.1. Leveraging the agility of ReWire's semantics-directed approach, we then derive two variants of the processor. The first variant, described in Sec. 4.2, implements a single-core processor with hardware-level support for separation. This processor supports interleaved computation in two security domains running on a shared execution unit. The second variant, described in Sec. 4.3, implements a dual-core processor with hardware-level separation. Our dual-core processor also features hardware-level support for separation, with each core dedicated to computation in a particular security domain.

4.1. ISA Specification

The instruction set architecture of the processor is borrowed from the PicoBlaze 8-bit soft microcontroller from Xilinx [Xilinx 2011], 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. 6) is often acceptable in




```
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 }
```

Fig. 3. (left) 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. (right) The types `Inputs` and `Outputs` for the ReWire version of PicoBlaze. Here, types of the form `W`n refer to n-bit words.

exchange for the expressive power, extensibility, and ease of formal verification that ReWire provides.

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 [Xilinx 2011]. 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 3 (left) 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.

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 higher-order functions. For synthesis, this will require a straightforward program transformation called defunctionalization [Reynolds 1972], 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 [Procter et al. 2015a], begins with preliminary type definitions. First, we define a reactive monad for the instruction set semantics called `ISAM` (for *ISA monad*).

```
type ISAM = ReT Inputs Outputs (StT ISAState I)
```

In other words, the PicoBlaze instruction set semantics lives in a reactive monad with inputs/outputs of type `Inputs` and `Outputs`, and a mutable internal state of type `ISAState`. The `Input` and `Output` types are defined in Fig. 3 (right) as record types, corresponding exactly to the block diagram of Fig. 3 (left) except for the clock signal (which is always implicitly present in ReWire). 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 ISAState = ISAState {
 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 and 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, is a 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; full code is available in the online supplement [Procter et al. 2015a].

4.1.3. Main Loop and Startup. The processor's execution is structured as a loop whose basic form is as follows:

```
loop :: Inputs -> ISAM ()
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
     Binop ...   -> ⟨handle binop instruction⟩              (†)
     Branch ...  -> ⟨handle branch instruction⟩
        ...
     Output ...  -> ⟨handle output instruction⟩
     Invalid ... -> ⟨handle invalid instruction⟩
```

where each of the elided codepaths ultimately results in a guarded tail call back to `loop`, with the incoming input signal passed as a parameter to the next loop iteration.

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 use a dual-port block RAM for the register file; we may therefore fetch both the registers `rx` and `ry` in one clock cycle.

```
Binop 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 initial output of all zeros, and enters the `loop`.

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

4.2. Separating Single-Core Processor

We now demonstrate how the ISA specification of Sec. 4.1 may be reused to implement a single-core processor with a shared execution unit that supports interleaved computation on two separate input streams. In particular, we will extend the design of Sec. 4.1 by wrapping the instantiated processor core with a *security harness* whose design is akin to a software-based monadic separation kernel [Harrison and Hook 2009]. The construction of the harness ensures non-interference, i.e. that low-security outputs will never be influenced by high-security inputs.

4.2.1. The Security Harness. The security harness serves to "lift" the action of the processor core into a *layered state monad* [Harrison and Hook 2009]. Use of a layered state monad enables precise control of cross-domain information flow. For the single-core harness we will provide two layers of state: one for the high core's internal state, and one for the low core's internal state. We will modify the input and output channels for the system to use the `Either` type, defined in Haskell as follows.

```
data Either a b = Left a | Right b
```

Input and output values tagged with `Left` correspond to the low security domain, while values tagged with `Right` correspond to the high security domain. Thus we arrive at the secure single-core monad `SCM`, where the harness lives:

```
type SCM = ReT (Either Inputs Inputs) (Either Outputs Outputs) K
type K   = StT ISAState (StT ISAState I)
```

The `harness` will operate in a tail recursive fashion, taking two `ISAM` computations reflecting the current execution state of the high and low domains respectively, and producing a computation in `SCM`. (Note that the definition of `harness` utilizes a number of helper functions that will be explained below.) When an input arrives, the harness examines the constructor tag (`Left` or `Right`) and executes one step in either the low or the high security domain. 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, tagged with `Left` or `Right` as needed. The harness then feeds the filtered inputs to the core and returns (via tail recursion) to the top of the loop.

```
harness :: Either Inputs Inputs
             -> ISAM a -> ISAM b
           -> SCM (Either a b)
harness (Left i) lo hi  = do
  r <- lift (liftKL (deReacT lo))
  case r of
    Left a      -> return (Left a)
    Right (o,k) -> do i' <- signal (Left o)
                      harness i' (k i) hi
harness (Right i) lo hi = do
  r <- lift (liftKH (deReacT hi))
  case r of
    Left a      -> return (Right a)
    Right (o,k) -> do i' <- signal (Right o)
                      harness i' lo (k i)
```

Helper functions `liftKL` and `liftKH` allow state actions of the individual cores to be mapped onto their respective state domains in the layered monad `SCM`. They are defined as follows.

```
liftKL :: StT ISAState I a -> K a
liftKL = lift

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

The addition of this security harness is all that is needed to implement the secure single-core processor. Section 6 touches on the performance characteristics of the circuit generated by ReWire.

4.3. Separating Dual-Core Processor

Producing a secure *dual*-core processor that implements a "no-write-down" policy is nearly as easy as the secure single-core. This time, we extend the design of Sec. 4.1 by instantiating *two* processor cores, and wrapping them with a security harness very similar to that of the preceding section. One of these cores will be designated as the "high" core and the other as the "low" core, reflecting different levels in a security lattice. To make things more interesting, 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.

4.3.1. The Dual-Core Harness. 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 of the dual-core processor are pairs. The dual-core monad `DCM`, then, is as follows:

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

As before, the `harness` will take as arguments two `ISAM` computations reflecting the current execution state of the high and low cores respectively, this time producing a computation in `DCM`. Unlike before, the harness loop proceeds by running *both* cores in parallel for a single step against their respective state layers. (Definitions of `liftKL` and `liftKH` are analogous to before, but must lift through one more state layer due to the presence of the shared register.) 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 :: ISAM a -> ISAM 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')
```

(One might reasonably ask how we can be certain that the low and high cores are executing in parallel rather than serially, as would seem to be suggested by the sequential flavor of the code. The essential answer is that no data dependencies can exist between the monadic actions for `r_lo` and `r_hi`, as these take place at different state monad layers. The performance results of Sec. 6 demonstrate that the FPGA synthesis tools are clever enough to notice this fact.)

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 writes to 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. In Sec. 5 we shall demonstrate the ease with which this processor's security may be verified, and Sec. 6 discusses the performance characteristics of the circuit generated by ReWire.

5. VERIFICATION

Owing to space constraints, we will restrict our attention here to the dual-core processor, though the security argument for the single-core processor is similar. To specify the security of the processor harness, we apply a security model developed for modular monadic semantics called *take separation* [Harrison and Hook 2009]. With this approach, the operation of (`harness lo hi`) is compared to the operation of (`harness lo skip`), where `skip` is a "no-op" 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 "no-op" core:

```
skip :: Outputs -> Inputs -> ReT Inputs Outputs K a
skip o i = signal 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 5.1 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 [Goguen and Meseguer 1990].

THEOREM 5.1 (HARNESS SECURITY). *For any appropriately typed* $\mathtt{i}_0$, $\mathtt{o}_0$, *os, finite is, lo and hi,*

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

PROOF. See appendix. □

Note that our security property is strong enough that it precludes not only storage channels, but also timing channels (assuming time is observable at clock-cycle granularity) and control flow channels.

6. SYNTHESIS

We now turn our attention briefly to the performance characteristics of the circuits synthesized from the specifications of Sec. 4. For benchmarking purposes, we will consider four separate artifacts: (1) "original" PicoBlaze (PB_O) as implemented by Xilinx in VHDL; (2) "plain" PicoBlaze (PB) as implemented in ReWire (Sec. 4.1); (3) "secure single-core" PicoBlaze (SPB_S) as implemented in ReWire (Sec. 4.2); and (4) "secure dual-core" PicoBlaze (SPB_D) as implemented in ReWire (Sec. 4.3). In Sec. 6.1, we compare PB_O with PB to obtain a sense of how much overhead is imposed by the use of ReWire as a source language. In Sec. 6.2, we compare SPB_S and SPB_D to PB to demonstrate that performance scales as expected in this example.

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 called *defunctionalization* [Reynolds 1972] that allows us to do just this in a straightforward, mechanical way. In this example, the outcome 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 will be replaced with calls to an interpretation function. This suffices to produce a program that is compilable by ReWire.

6.1. Comparison with Original Implementation

To evaluate performance, both our PB and Xilinx's PB_O 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)
PB_O	99	76	181	139.919
PB	398	110	763	64.956

Put another way, the ReWire-based processor is approximately 4.0 times as large as the original (as measured in slices), and is capable of operating at about 46.4% the maximum clock speed. We believe these performance results, being 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.

6.2. Performance of the Secure Processors

The secure processors make much more extensive use of higher-order language features than the basic single-core processor. In particular, the harness loops take two monadic computations as arguments. Nevertheless, defunctionalization still suffices to transform the harness-based specifications into a first-order, compilable form. The fully defunctionalized version of the processors and harnesses are available in the code repository [Procter et al. 2015a].

Synthesis results. Synthesis estimates for device utilization and maximum clock speed of the secure single- and dual-core processors were obtained by synthesizing the ReWire-implemented portions of the three devices (the non-separating processor, the separating single-core processor, and the separating dual-core processor) with the same XST settings as in Sec. 6.1, then taking the ratio of occupied logic slices, flip flops, LUTs, and F_{max} for the synthesized secure processors to the corresponding results for the non-separating processor. The following table illustrates the results.

	Slices	Flip Flops	4-LUTs	F_{max}
Ratio of SPB_D to PB	2.026	2.283	2.033	0.950
Ratio of SPB_S to PB	1.264	2.228	1.259	0.893

While the experiment here is somewhat modest in scope, considering only single- a dual-core processors rather than larger multi-core processors (say, quad- or octa-core), the result does suggest reasonable scaling with respect to area and performance. For the dual-core processor, slice and LUT utilization are almost exactly twice as much as the basic single core processor (as would be expected in the presence of two processor cores), while flip flop utilization suffers a slight extra 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 5% of the basic single-core processor. For the single-core processor, we pay a slightly higher penalty with respect to logic utilization, likely attributable

to the more complex routing logic for I/O signals. This contributes to an F_{max} value that is slightly lower, around 10% slower than the original ReWire-based single-core processor.

7. CONCLUSIONS AND FUTURE WORK

This article 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 this article demonstrates, this design paradigm brings great benefits with respect to the modular construction and formal verification of hardware.

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. A graphical front-end for ReWire is currently in development [Graves 2015] that will aid hardware engineers and encourage adoption of the ReWire tools.

It would be illuminating to extend the case study of Sections 4–6 in two respects: first, with respect to circuit scale (i.e., number of cores), and second, with respect to the class of attacks addressed (considering, for example, timing channels introduced by a shared cache). With respect to the latter, we believe that ReWire would be an excellent framework for verifying security particularly with respect to timing channels induced by cache misses. At a high level, we envision constructing a cache module that tags each cache line with an additional "knowledge bit" that indicates whether the low-security core(s) should know that that particular location is in the cache. If this bit is not set, the cache would return a "false miss" to any low security core requesting that location. Once the resulting memory request from the low core is serviced, the knowledge bit in the corresponding cache line can then be set, permitting subsequent requests by low cores for that location to be serviced from the cache. The resulting system would allow the sharing of cache lines between high and low cores, and yet, from the low cores' perspective, exhibit the same timing properties regardless of any action taken by the high-security core(s). Verifying security of this system would amount to proving a trace property that is conceptually very similar to, if somewhat more complex than, the property verified in Section 5. We intend to explore this exact example in future work. By contrast, we note in passing that ReWire is not well suited to working with physical side-channel attacks induced by, for example, monitoring of power consumption; ReWire is intended only to model circuit behavior at the clock-cycle level. On the other hand, security properties relating to integrity (as opposed to confidentiality) should be straightforward to verify, as these are essentially dual to the kinds of information flow property we have examined in the present work.

When translating from verified high-level specifications to hardware-level implementations, the correctness of the toolchain is clearly critical to the overall verification argument. In a sense, this is the entire point of language-based formal methods: proof of the adequacy of our high level models may be reduced to a proof of compiler correctness! In that spirit, work is ongoing to verify the correctness of the ReWire toolset. Pre-

liminary results include an encoding of ReWire's formal semantics in the Coq theorem prover, to be discussed in a forthcoming publication. Mechanizing ReWire's metatheory will come with an additional benefit where verification is concerned: the formal development of its semantics also includes a formalization of the equational logic used in this paper. While we believe that the equational logic provided by ReWire is already easy to work with on paper, a mechanized proof framework will greatly increase the productivity of formal methods practitioners.

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 [Taha and Sheard 2000]) 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 [Harrison et al. 2012]; 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 non-tail recursion at runtime—is not synthesizable by 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 will provide a powerful tool for tackling the coordination challenges inherent to such heterogeneous systems.

REFERENCES

Christiaan Baaij and Jan Kuper. 2014. Using Rewriting to Synthesize Functional Languages to Digital Circuits. In *Proceedings of the 2014 International Symposium on Trends in Functional Programming (TFP'14)*. 17–33.

J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. In *2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1212–1221.

Per Bjesse, Koen Claessen, and Mary Sheeran. 1998. Lava: Hardware Design in Haskell. In *ICFP '98*. 174–184.

David Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*. 167–182.

Stephen A. Edwards. 2006. The Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design and Test of Computers* 23, 5 (2006), 375–386.

Anthony Fox and Magnus O. Myreen. 2010. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the First International Conference on Interactive Theorem Proving (ITP'10)*. 243–258.

Nithin George, Hyoukjoong Lee, David Novo, Tiark Rompf, Kevin Brown, Arvind Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware System Synthesis from Domain-Specific Languages. In *Proceedings of 24th Internation Conference on Field Programmable Logic and Applications (FPL '14)*.

Andy Gill. 2011. Declarative FPGA Circuit Synthesis using Kansas Lava. In *ERSA '11*.

Andy Gill. 2014. Domain-specific Languages and Code Synthesis Using Haskell. *ACM Queue* 12, 4, Article 30 (April 2014), 14 pages.

Carlos Eduardo Giménez. 1996. *Un Calcul De Constructions Infinies Et Son Application A La Verification De Systemes Communicants*. Ph.D. Dissertation. L'École Normale Supérieure de Lyon.

Joseph A. Goguen and José Meseguer. 1990. Security Policies and Security Models. In *Proceedings of the 1982 Symposium on Security and Privacy (SSP '82)*. IEEE Computer Society Press, 11–20.

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

Ian Graves. 2015. *Device-Level Composition in ReWire*. Ph.D. Dissertation. University of Missouri.

W. Harrison, A. Procter, J. Agron, G. Kimmel, and G. Allwein. 2009. Model-Driven Engineering from Modular Monadic Semantics: Implementation Techniques Targeting Hardware and Software. In *DSL '09: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*. 20–44.

W. L. Harrison. 2006. Proof Abstraction for Imperative Languages. In *Proceedings of the 4th Asian Conference on Programming Languages and Systems (APLAS'06)*. 97–113.

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

William L. Harrison and Adam Procter. 2015. Cheap (But Functional) Threads. (2015). 44 pages. Accepted for publication in *Higher-Order and Symbolic Computation*.

William L. Harrison, Adam Procter, and Gerard Allwein. 2012. The confinement problem in the presence of faults. In *Proceedings of the 14th International Conference on Formal Engineering Methods (ICFEM'12)*. 182–197.

HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro* 31, 5 (Sept. 2011), 42–53.

Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. 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)*. ACM, New York, NY, USA, 109–120.

Sheng Liang. 1998. *Modular Monadic Semantics and Compilation*. Ph.D. Dissertation. Yale University.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*.

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

Gerald J. Popek and Robert P. Goldberg. 1974. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM* 17, 7 (July 1974), 412–421.

Adam Procter. 2014. *Semantics-Driven Design and Implementation of High-Assurance Hardware*. Ph.D. Dissertation. University of Missouri.

Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. 2015a. Online supplement accompanying "A Principled Approach to Secure Multi-Core Processor Design in ReWire". (2015). http://adamprocter.com/tecs15

Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. 2015b. Semantics Driven Hardware Design, Implementation, and Verification with ReWire. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'15)*. Article 13, 13:1–13:10 pages.

John Reynolds. 1972. Definitional Interpreters for Higher Order Programming Languages. *ACM Conference Proceedings* (1972), 717–740.

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

Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. 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)*. 379–391.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 12 (2000), 211–242.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of Computer Security* 4, 2–3 (1996), 167–187.

Matthew Wilding, David Greve, Raymond Richards, and David Hardin. 2010. Formal Verification of Partition Management for the AAMP7G Microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, David S. Hardin (Ed.). 175–191.

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

Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. 2015. Hardware Synthesis from a Recursive Functional Language. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

A. SECURITY VERIFICATION

The proof of Theorem 5.1 is by induction on the length of `is`, which is assumed to be finite, and uses the techniques established in previous work [Harrison 2006; Harrison and Hook 2009] in which by-contruction properties of operations on layered state monads (e.g., `K`) are used to prove the equality. The three principal properties used are *atomic noninterference*, computational innocence, and the *clobber* rule. We describe these properties informally as the technical details may be found in the aforementioned articles.

A layered state monad is a monad constructed from multiple applications of the state monad transformer. The monad `K`, for example, is the result of three applications of state monad transformer to the identity monad:

```
type K = (StT SharedReg
          (StT ISAState
           (StT ISAState I)))
```

Atomic noninterference formalizes the notion that operations (i.e., atoms) lifted from distinct layers in a layered commute (i.e., do not interfere) with the monadic bind operator. Computational innocence shows how computations that are side-effect free (i.e., "innocent" computations) may be added to other computations preserving equality. For example, for the `get` operation defined by `StT`, $\texttt{get} >> \varphi = \varphi$ for any computation φ. Finally, the "clobber rule" shows that operations within the same state layer may be cancelled out—i.e., clobbered. For example in `K`, we defined $\texttt{mask}_\texttt{H}$ as:

```
maskH :: K ()
maskH = liftKH (update (const s0))
   where s0 = undefined
```

By the clobber rule, $\texttt{liftKH}\,\varphi >> \texttt{maskH} = \texttt{maskH} = \texttt{liftKH}\,\gamma >> \texttt{maskH}$ for any appropriately typed φ and γ.

Additionally, the "monad laws" [Liang 1998] are also applied extensively throughout the verification. These are:

$\texttt{return v} >>= \texttt{f} = \texttt{f v}$ — left unit

$\texttt{x} >>= \texttt{return} = \texttt{x}$ — right unit

$(\texttt{x} >>= \lambda\texttt{v}.\,\texttt{y}) >>= \lambda\texttt{w}.\,\texttt{z} = \texttt{x} >>= \lambda\texttt{v}.\,(\texttt{y} >>= \lambda\texttt{w}.\,\texttt{z})$ — associativity

The proof of Theorem 5.1 follows the pattern, illustrated below. In the informal sketch below, we do some violence to the syntax in order to provide the reader with a roadmap to the proof of Theorem 5.1. The first step unrolls the operation of $(\texttt{harness lo hi})$ into a sequence of operations, lh_i, which combine actions from both `lo` and `hi` and their operations on the shared register layer. The idempotence of $\texttt{mask}_\texttt{H}$ is used to clone it and associativity is used to move $\texttt{mask}_\texttt{H}$ to the right of lh_n. The clobber rule is used to cancel `hi`'s actions, producing l_n whose actions consist only of `lo`'s and `lo`'s writes to the shared register. $\texttt{mask}_\texttt{H}$ commutes with $\texttt{l}_\texttt{n}$ and this clobber-then-commute pattern is repeated until all of `hi`'s effects have been cancelled. Then, the cloned $\texttt{mask}_\texttt{H}$ may be "backed out" and removed by its idempotence. The result is equal to the r.h.s. of Theorem 5.1.

$pull\ os\ [i_1, \ldots, i_n]\ (harness\ lo\ hi)\ >>= \lambda os.\ mask_H >>\ \texttt{return}\ os$

$= (\texttt{lh}_1\,; \ldots ;\,\texttt{lh}_\texttt{n})\ >>= \lambda\texttt{os}.\,\texttt{mask}_\texttt{H} >>\ \texttt{return os}$ — $mask_H$ idempotent

$= (\texttt{lh}_1\,; \ldots ;\,\texttt{lh}_\texttt{n}\,;\,\texttt{mask}_\texttt{H})\ >>= \lambda\texttt{os}.\,\texttt{mask}_\texttt{H} >>\ \texttt{return os}$ — assoc.

$= (\texttt{lh}_1\,; \ldots ;\,\texttt{l}_\texttt{n}\,;\,\texttt{mask}_\texttt{H})\ >>= \lambda\texttt{os}.\,\texttt{mask}_\texttt{H} >>\ \texttt{return os}$ — clobber

$= (\texttt{lh}_1\,; \ldots ;\,\texttt{mask}_\texttt{H}\,;\,\texttt{l}_\texttt{n})\ >>= \lambda\texttt{os}.\,\texttt{mask}_\texttt{H} >>\ \texttt{return os}$ — atomic nonint.

$= (\texttt{lh}_1\,;\,\texttt{mask}_\texttt{H}\,; \ldots ;\,\texttt{l}_\texttt{n})\ >>= \lambda\texttt{os}.\,\texttt{mask}_\texttt{H} >>\ \texttt{return os}$ — atomic nonint.

$= (\texttt{l}_1\,;\,\texttt{mask}_\texttt{H}\,; \ldots ;\,\texttt{l}_\texttt{n})\ >>= \lambda\texttt{os}.\,\texttt{mask}_\texttt{H} >>\ \texttt{return os}$ — clobber

$= (\mathtt{l}_1\ ;\ \ldots ;\ \mathtt{l}_\mathtt{n})\ \gg\!=\ \lambda \mathtt{os}.\ \mathtt{mask}_\mathtt{H}\ \gg\ \mathtt{return}\ \mathtt{os}$ — "reversing previous steps"
$= \mathit{pull}\ \mathit{os}\ [i_1, \ldots, i_n]\ (\mathit{harness}\ \mathit{lo}\ (\mathit{skip}\, o_0\, i_0))\ \gg\!=\ \lambda \mathit{os}.\ \mathit{mask}_H\ \gg\ \mathtt{return}\ \mathit{os}$

The remainder of this appendix consists of the following. Section A.1 discusses lemmas which simplify the proof of Theorem 5.1. These lemmas follow by routine, if somewhat laborious, application and simplification of the definitions of the harness. We include the proof of Lemma A.4 which is the most complex of the lemmas. Section A.2 contains the proof of Theorem 5.1. Section B presents the proof of Lemma A.4.

A.1. Lemmas

This section presents four lemmas used to prove Theorem 5.1. Each of them involves unfolding definitions from the harness and simplifying using the monad laws, β-reduction, etc. The proof of Lemma A.4 is presented in Section B.

Lemma A.1 unwinds the definition of `pull` on an n length input list into n calls to `next`.

LEMMA A.1 (PULL). *Given φ and os of appropriate type. For every* $\mathtt{n} \in \mathbb{N}$*,*

$$\begin{aligned}\mathit{pull}\ \mathit{os}\ [i_1, \ldots, i_n]\ \varphi = \ & \mathit{next}\ \varphi && \gg\!=\ \lambda \mathit{Right}\ (o_1, \kappa_1).\\ & \mathit{next}\ (\kappa_1\ i_1) && \gg\!=\ \lambda \mathit{Right}\ (o_2, \kappa_2).\\ & \qquad\vdots\\ & \mathit{next}\ (\kappa_{n-1}\ i_{n-1}) && \gg\!=\ \lambda \mathit{Right}\ (o_n, \kappa_n).\\ & \mathtt{return}\ (\mathit{os} +\!\!+ [\mathit{fst}\ o_1, \ldots, \mathit{fst}\ o_n])\end{aligned}$$

□

Lemma A.2 formulates the interaction of `next` with `harness`.

LEMMA A.2 ($\mathit{next} \circ \mathit{harness}$). *For any appropriately typed* `hi` *and* `lo`

$$\begin{aligned}&\mathit{next}\ (\mathit{harness}\ \mathit{lo}\ \mathit{hi})\\ &= (\mathit{lift}\ .\ \mathit{lift}^l_K)\ (\mathit{next}\ \mathit{lo}) \ \gg\!=\ \lambda \mathit{Right}\ (o^l, \kappa^l).\\ &\quad (\mathit{lift}\ .\ \mathit{lift}^h_K)\ (\mathit{next}\ \mathit{hi}) \ \gg\!=\ \lambda \mathit{Right}\ (o^h, \kappa^h)\\ &\quad \mathbf{let}\\ &\qquad \mathtt{f} = \lambda(\mathtt{i}^\mathtt{l}, \mathtt{i}^\mathtt{h}).\ \mathit{checkHiPort}\ i^h\ o^h \ \gg\!=\ \lambda\hat{i}^h.\\ &\qquad\qquad \mathit{checkLoPort}\ o^l \ \gg\\ &\qquad\qquad \mathit{harness}\ (\kappa^l\ i^l)\ (\kappa^h\ \hat{i}^h)\\ &\quad \mathbf{in}\\ &\quad \mathtt{return}\ (\mathtt{Right}\ ((\mathtt{o}^\mathtt{l},\ \mathtt{o}^\mathtt{h}),\ \mathtt{f}))\end{aligned}$$

□

Lemma A.3 formulates the interaction between `next` and the `lift` for the `ReT` monad transformer. N.b., `next` behaves as a kind of inverse or project for that `lift`.

LEMMA A.3 (NEXT ∘ LIFT). *The following holds.*

$$\mathtt{next}\ (\mathtt{lift}\ \mathtt{x} \ \gg\!=\ \mathtt{f}) = \mathtt{x} \ \gg\!=\ \mathtt{next}\ .\ \mathtt{f}$$

□

Lemma A.4 captures the interaction of `pull` with `harness` in which a call to `pull` on `harness` is reduced to a (co)recursive call.

LEMMA A.4 (PULL ∘ HARNESS). *For appropriately typed os, hi and lo, and assuming WLOG that $i_1 = (i^l_1, i^h_1)$,*

$$
\begin{array}{l}
\mathit{pull}\ \mathit{os}\ [i_1,\ldots,i_n]\ (\mathit{harness}\ \mathit{lo}\ \mathit{hi}) \\
\quad = \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) \quad \gg\!= \quad \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\qquad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) \quad \gg\!= \quad \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\qquad \mathit{chkHPrt}\ \ i^h_1\ \ o^h_1 \quad \gg\!= \quad \lambda \hat{i}^h. \\
\qquad \mathit{chkLPrt}\ \ o^l_1 \qquad \gg\!= \\
\qquad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2,\ldots,i_n]\ \ (\kappa^l\ \ i^l_1)\ (\kappa^h\ \hat{i}^h)
\end{array}
$$

□

A.2. Theorem 5.1

PROOF. Proof of Theorem 5.1.

$$
\begin{array}{l}
\mathit{pull}\ \mathit{os}\ ((i^l_1, i^h_1) : \mathit{is})\ (\mathit{harness}\ \mathit{lo}\ \mathit{hi}) \ \gg\!= \lambda v.\ \mathit{mask}_H \gg\!\!\gg \texttt{return}\ v \\
\{\mathit{Lemma\ A.4.}\} \\
\quad = \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) \qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\qquad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) \qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\qquad \mathit{chkHPrt}\ \ i^h\ \ o^h_1 \qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \hat{i}^h. \\
\qquad \mathit{chkLPrt}\ \ o^l_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \gg\!\!\gg \\
\qquad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2,\ldots,i_n]\ \ (\kappa^l\ \ i^l)\ (\kappa^h\ \hat{i}^h) \quad \gg\!= \quad \lambda v.\ \mathit{mask}_H \gg\!\!\gg \texttt{return}\ v \\
\{\mathit{Induction\ hypothesis.}\} \\
\quad = \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\qquad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\qquad \mathit{chkHPrt}\ \ i^h\ \ o^h_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \hat{i}^h. \\
\qquad \mathit{chkLPrt}\ \ o^l_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \gg\!\!\gg \\
\qquad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2,\ldots,i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ \hat{i}^h) \quad \gg\!= \quad \lambda v.\ \mathit{mask}_H \gg\!\!\gg \texttt{return}\ v \\
\{\mathit{Defn.\ skip.}\} \\
\quad = \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\qquad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\qquad \mathit{chkHPrt}\ \ i^h\ \ o^h_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \hat{i}^h. \\
\qquad \mathit{chkLPrt}\ \ o^l_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \gg\!\!\gg \\
\qquad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2,\ldots,i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) \quad \gg\!= \quad \lambda v.\ \mathit{mask}_H \gg\!\!\gg \texttt{return}\ v \\
\{\mathit{Defn.\ chkHPrt,\ innocence.}\} \\
\quad = \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\qquad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\qquad \mathit{chkLPrt}\ \ o^l_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \gg\!\!\gg \\
\qquad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2,\ldots,i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) \quad \gg\!= \quad \lambda v.\ \mathit{mask}_H \gg\!\!\gg \texttt{return}\ v \\
\{\mathit{Defn.}\ \gg\!\!\gg;\ o^h_1, \kappa^h\ \mathit{free.}\} \\
\quad = \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\qquad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \gg\!\!\gg \\
\qquad \mathit{chkLPrt}\ \ o^l_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \gg\!\!\gg \\
\qquad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2,\ldots,i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) \quad \gg\!= \quad \lambda v.\ \mathit{mask}_H \gg\!\!\gg \texttt{return}\ v \\
\{\mathit{mask}_H\ \mathit{idempotent.}\} \\
\quad = \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= \quad \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\qquad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \gg\!\!\gg \\
\qquad \mathit{chkLPrt}\ \ o^l_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \gg\!\!\gg \\
\qquad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2,\ldots,i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) \quad \gg\!= \quad \lambda v.\ \mathit{mask}_H \gg\!\!\gg \mathit{mask}_H \gg\!\!\gg \texttt{return}\ v
\end{array}
$$

$$
\begin{array}{lll}
\{\textit{Consequence of atomic noninterference.}\} & & \\
\quad = \; \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) & \texttt{>>=} & \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\quad\quad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) & \texttt{>>} & \\
\quad\quad \mathit{chkLPrt}\ \ o^l_1 & \texttt{>>} & \\
\quad\quad \mathit{mask}_H & \texttt{>>} & \\
\quad\quad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) & \texttt{>>=} & \lambda v.\ \mathit{mask}_H \texttt{>>}\ \texttt{return}\ v \\
\{\textit{Consequence of atomic noninterference.}\} & & \\
\quad = \; \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) & \texttt{>>=} & \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\quad\quad \mathit{lift}^h_K\ (\mathit{next}\ \mathit{hi}) & \texttt{>>} & \\
\quad\quad \mathit{mask}_H & \texttt{>>} & \\
\quad\quad \mathit{chkLPrt}\ \ o^l_1 & \texttt{>>} & \\
\quad\quad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) & \texttt{>>=} & \lambda v.\ \mathit{mask}_H \texttt{>>}\ \texttt{return}\ v \\
\{\textit{Consequence of clobber.}\} & & \\
\quad = \; \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) & \texttt{>>=} & \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\quad\quad \mathit{lift}^h_K\ (\mathit{next}\ (\mathit{skip}\ o_0\ i_0)) & \texttt{>>} & \\
\quad\quad \mathit{mask}_H & \texttt{>>} & \\
\quad\quad \mathit{chkLPrt}\ \ o^l_1 & \texttt{>>} & \\
\quad\quad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) & \texttt{>>=} & \lambda v.\ \mathit{mask}_H \texttt{>>}\ \texttt{return}\ v \\
\{\textit{Reversing previous steps.}\} & & \\
\quad = \; \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) & \texttt{>>=} & \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\quad\quad \mathit{lift}^h_K\ (\mathit{next}\ (\mathit{skip}\ o_0\ i_0)) & \texttt{>>} & \\
\quad\quad \mathit{chkLPrt}\ \ o^l_1 & \texttt{>>} & \\
\quad\quad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) & \texttt{>>=} & \lambda v.\ \mathit{mask}_H \texttt{>>}\ \texttt{return}\ v \\
\{\textit{Defn.}\ \texttt{>>};\ o^h_1, \kappa^h\ \textit{free.}\} & & \\
\quad = \; \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) & \texttt{>>=} & \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\quad\quad \mathit{lift}^h_K\ (\mathit{next}\ (\mathit{skip}\ o_0\ i_0)) & \texttt{>>=} & \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\quad\quad \mathit{chkLPrt}\ \ o^l_1 & \texttt{>>} & \\
\quad\quad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) & \texttt{>>=} & \lambda v.\ \mathit{mask}_H \texttt{>>}\ \texttt{return}\ v \\
\{\textit{Consequence of innocence.}\} & & \\
\quad = \; \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) & \texttt{>>=} & \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\quad\quad \mathit{lift}^h_K\ (\mathit{next}\ (\mathit{skip}\ o_0\ i_0)) & \texttt{>>=} & \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\quad\quad \mathit{chkHPrt}\ \ i^h\ \ o^h_1 & \texttt{>>=} & \lambda \hat{i}^h. \\
\quad\quad \mathit{chkLPrt}\ \ o^l_1 & \texttt{>>} & \\
\quad\quad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ i_0) & \texttt{>>=} & \lambda v.\ \mathit{mask}_H \texttt{>>}\ \texttt{return}\ v \\
\{\textit{Defn. of skip.}\} & & \\
\quad = \; \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) & \texttt{>>=} & \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\quad\quad \mathit{lift}^h_K\ (\mathit{next}\ (\mathit{skip}\ o_0\ i_0)) & \texttt{>>=} & \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\quad\quad \mathit{chkHPrt}\ \ i^h\ \ o^h_1 & \texttt{>>=} & \lambda \hat{i}^h. \\
\quad\quad \mathit{chkLPrt}\ \ o^l_1 & \texttt{>>} & \\
\quad\quad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\mathit{skip}\ o_0\ \hat{i}^h) & \texttt{>>=} & \lambda v.\ \mathit{mask}_H \texttt{>>}\ \texttt{return}\ v \\
\{\textit{Defn. skip, next};\ \texttt{return}\ v\ \texttt{>>=}\ \lambda x.e\ =\ \texttt{return}\ v\ \texttt{>>=}\ \lambda x.e[x/v].\} & & \\
\quad = \; \mathit{lift}^l_K\ (\mathit{next}\ \mathit{lo}) & \texttt{>>=} & \lambda \mathit{Right}\ (o^l_1, \kappa^l). \\
\quad\quad \mathit{lift}^h_K\ (\mathit{next}\ (\mathit{skip}\ o_0\ i_0)) & \texttt{>>=} & \lambda \mathit{Right}\ (o^h_1, \kappa^h). \\
\quad\quad \mathit{chkHPrt}\ \ i^h\ \ o^h_1 & \texttt{>>=} & \lambda \hat{i}^h. \\
\quad\quad \mathit{chkLPrt}\ \ o^l_1 & \texttt{>>} & \\
\quad\quad \mathit{pull}\ \ (\mathit{os} +\!\!+ [o^l_1])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\kappa^h\ \hat{i}^h) & \texttt{>>=} & \lambda v.\ \mathit{mask}_H \texttt{>>}\ \texttt{return}\ v
\end{array}
$$

$\{Lemma\ A.4.\}$
$= \texttt{pull os } ((\texttt{i}^{\texttt{l}}_1, \texttt{i}^{\texttt{h}}_1) : \texttt{is}) \ (\texttt{harness lo } (\texttt{skip } \texttt{o}_0 \ \texttt{i}_0)) \ \texttt{>>=} \ \lambda \texttt{v}.\ mask_{\texttt{H}} \ \texttt{>>} \ \texttt{return v}$

□

B. LEMMA A.4 PROOF

PROOF. Lemma A.4.

$$pull\ os\ [i_1, \ldots, l_n]\ (harness\ lo\ hi)$$

$\{Lemma\ A.1.\}$

$$\begin{aligned}
= \ & next\ (harness\ lo\ hi) && \texttt{>>=} \ \lambda Right(o_1, \kappa_1). \\
& next\ (\kappa_1\ i_1) && \texttt{>>=} \ \lambda Right(o_2, \kappa_2). \\
& && \vdots \\
& next\ (\kappa_{n-1}\ i_{n-1}) && \texttt{>>=} \ \lambda Right(o_n, \kappa_n). \\
& \texttt{return}(os \ +\!\!+ \ [o_1, \ldots, o_n])
\end{aligned}$$

$\{Lemma\ A.2.\}$

$$\begin{aligned}
= \ & next \left(\begin{array}{l}
(lift\ .\ lift^l_K)\ (next\ lo) \quad \texttt{>>=} \ \lambda Right\ (o^l, \kappa^l). \\
(lift\ .\ lift^h_K)\ (next\ hi) \quad \texttt{>>=} \ \lambda Right\ (o^h, \kappa^h) \\
\textbf{let} \\
\quad \texttt{f} \ = \ \lambda(\texttt{i}^{\texttt{l}}, \texttt{i}^{\texttt{h}}).\ checkHiPort\ \ i^h\ \ o^h \quad \texttt{>>=} \ \lambda \hat{i}^h. \\
\qquad\qquad\qquad\quad checkLoPort\ \ o^l \quad \texttt{>>} \\
\qquad\qquad\qquad\quad harness\ (\kappa^l\ \ i^l)\ (\kappa^h\ \hat{i}^h) \\
\textbf{in} \\
\texttt{return}\ (\texttt{Right}\ ((\texttt{o}^{\texttt{l}},\ \texttt{o}^{\texttt{h}}),\ \texttt{f}))
\end{array} \right) \ \texttt{>>=} \ \lambda Right(o_1, \kappa_1). \\
& next\ (\kappa_1\ i_1) \qquad\qquad\qquad \texttt{>>=} \ \lambda Right(o_2, \kappa_2). \\
& \qquad\qquad\qquad\qquad\qquad\quad \vdots \\
& next\ (\kappa_{n-1}\ i_{n-1}) \qquad\qquad \texttt{>>=} \ \lambda Right(o_n, \kappa_n). \\
& \texttt{return}(os \ +\!\!+ \ [o_1, \ldots, o_n])
\end{aligned}$$

$\{Associativity\ of\ \texttt{>>=},\ Lemma\ A.3,\ Simplification.\}$

$$\begin{aligned}
= \ & lift^l_K\ (next\ lo) \quad \texttt{>>=} \ \lambda Right\ (o^l, \kappa^l). \\
& lift^h_K\ (next\ hi) \quad \texttt{>>=} \ \lambda Right\ (o^h, \kappa^h) \\
& \textbf{let} \\
& \quad \texttt{f} \ = \ \lambda(\texttt{i}^{\texttt{l}}, \texttt{i}^{\texttt{h}}).\ checkHiPort\ \ i^h\ \ o^h \quad \texttt{>>=} \ \lambda \hat{i}^h. \\
& \qquad\qquad\qquad\quad checkLoPort\ \ o^l \quad \texttt{>>} \\
& \qquad\qquad\qquad\quad harness\ (\kappa^l\ \ i^l)\ (\kappa^h\ \hat{i}^h) \\
& \textbf{in} \\
& next\ (\texttt{return}\ (Right\ ((o^l,\ o^h),\ f))) \quad \texttt{>>=} \ \lambda Right(o_1, \kappa_1). \\
& next\ (\kappa_1\ i_1) \qquad\qquad\qquad\qquad\quad \texttt{>>=} \ \lambda Right(o_2, \kappa_2). \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
& next\ (\kappa_{n-1}\ i_{n-1}) \qquad\qquad\qquad \texttt{>>=} \ \lambda Right(o_n, \kappa_n). \\
& \texttt{return}(os \ +\!\!+ \ [o_1, \ldots, o_n])
\end{aligned}$$

$\{Lemma\ A.3, \texttt{return} = lift \circ \texttt{return}_K.\}$

$$
\begin{array}{ll}
= & lift^l_K\ (next\ lo) \ \ \texttt{>>=}\ \ \lambda Right\ (o^l, \kappa^l). \\
& lift^h_K\ (next\ hi) \ \ \texttt{>>=}\ \ \lambda Right\ (o^h, \kappa^h) \\
& \textbf{let} \\
& \quad \texttt{f} \ = \ \lambda(\texttt{i}^\texttt{l}, \texttt{i}^\texttt{h}).\ checkHiPort\ \ i^h\ \ o^h \ \ \texttt{>>=}\ \ \lambda\hat{i}^h. \\
& \qquad\qquad\qquad checkLoPort\ \ o^l \ \ \texttt{>>} \\
& \qquad\qquad\qquad harness\ (\kappa^l\ \ i^l)\ (\kappa^h\ \hat{i}^h) \\
& \textbf{in} \\
& \texttt{return}_K\ (Right\ ((o^l,\ o^h),\ f)) \ \ \texttt{>>=}\ \ \lambda Right(o_1, \kappa_1). \\
& next\ (\kappa_1\ i_1) \ \ \texttt{>>=}\ \ \lambda Right(o_2, \kappa_2). \\
& \qquad \vdots \\
& next\ (\kappa_{n-1}\ i_{n-1}) \ \ \texttt{>>=}\ \ \lambda Right(o_n, \kappa_n). \\
& \texttt{return}(os\ +\!\!+\ [o_1, \ldots, o_n])
\end{array}
$$

$\{Left\ unit.\}$

$$
\begin{array}{ll}
= & lift^l_K\ (next\ lo) \ \ \texttt{>>=}\ \ \lambda Right\ (o^l, \kappa^l). \\
& lift^h_K\ (next\ hi) \ \ \texttt{>>=}\ \ \lambda Right\ (o^h, \kappa^h). \\
& \textbf{let} \\
& \quad \texttt{f} \ = \ \lambda(\texttt{i}^\texttt{l}, \texttt{i}^\texttt{h}).\ checkHiPort\ \ i^h\ \ o^h \ \ \texttt{>>=}\ \ \lambda\hat{i}^h. \\
& \qquad\qquad\qquad checkLoPort\ \ o^l \ \ \texttt{>>} \\
& \qquad\qquad\qquad harness\ (\kappa^l\ \ i^l)\ (\kappa^h\ \hat{i}^h) \\
& \textbf{in} \\
& next\ (f\ i_1) \ \ \texttt{>>=}\ \ \lambda Right(o_2, \kappa_2). \\
& \qquad \vdots \\
& next\ (\kappa_{n-1}\ i_{n-1}) \ \ \texttt{>>=}\ \ \lambda Right(o_n, \kappa_n). \\
& \texttt{return}(os\ +\!\!+\ [(o^l,\ o^h)_1, \ldots, o_n])
\end{array}
$$

$\{Consequence\ of\ Lemma\ A.3.\}$

$$
\begin{array}{ll}
= & lift^l_K\ (next\ lo) \ \ \texttt{>>=}\ \ \lambda Right\ (o^l, \kappa^l). \\
& lift^h_K\ (next\ hi) \ \ \texttt{>>=}\ \ \lambda Right\ (o^h, \kappa^h). \\
& chkHPrt\ \ i^h\ \ o^h \ \ \texttt{>>=}\ \ \lambda\hat{i}^h. \\
& chkLPrt\ \ o^l \ \ \texttt{>>} \\
& \textbf{let} \\
& \quad \texttt{f} \ = \ \lambda(\texttt{i}^\texttt{l}, \texttt{i}^\texttt{h}).\ harness\ (\kappa^l\ \ i^l)\ (\kappa^h\ \hat{i}^h) \\
& \textbf{in} \\
& next\ (f\ i_1) \ \ \texttt{>>=}\ \ \lambda Right(o_2, \kappa_2). \\
& \qquad \vdots \\
& next\ (\kappa_{n-1}\ i_{n-1}) \ \ \texttt{>>=}\ \ \lambda Right(o_n, \kappa_n). \\
& \texttt{return}(os\ +\!\!+\ [(o^l,\ o^h)_1, \ldots, o_n])
\end{array}
$$

$\{Substitution\ of\ \textbf{let}\ binding, Lemma\ A.1.\}$

$$
\begin{array}{ll}
= & lift^l_K\ (next\ lo) \ \ \texttt{>>=}\ \ \lambda Right\ (o^l, \kappa^l). \\
& lift^h_K\ (next\ hi) \ \ \texttt{>>=}\ \ \lambda Right\ (o^h, \kappa^h). \\
& chkHPrt\ \ i^h\ \ o^h \ \ \texttt{>>=}\ \ \lambda\hat{i}^h. \\
& chkLPrt\ \ o^l \ \ \texttt{>>} \\
& pull\ \ (os\ +\!\!+\ [(o^l,\ o^h)_1, \ldots, o_n])\ \ [i_2, \ldots, i_n]\ \ (\kappa^l\ \ i^l)\ \ (\kappa^h\ \ \hat{i}^h))
\end{array}
$$

□