A Core Calculus for Secure Hardware: Its Formal Semantics and Proof System

Thomas N. Reynolds*, Adam Procter†, William L. Harrison*, Gerard Allwein‡
*Department of Computer Science, University of Missouri
†Intel Corporation, Hillsboro, Oregon, USA
‡US Naval Research Laboratory, Washington, DC

***Abstract*—Constructing high assurance, secure hardware remains a challenge, because to do so relies on both a verifiable means of hardware description and implementation. However, production hardware description languages (HDL) lack the formal underpinnings required by formal methods in security. Still, there is no such thing as high assurance systems without high assurance hardware. We present a core calculus of secure hardware description with its formal semantics, security type system and mechanization in Coq. This calculus is the core of the functional HDL, ReWire, shown in previous work to have useful applications in reconfigurable computing. This work supports a full-fledged, formal methodology for producing high assurance hardware.**

I. Introduction

ReWire is a functional hardware description language (HDL): it is a functional language—a subset of Haskell—from which circuits are synthesized automatically. Previous work has introduced ReWire's language design and implementation as well as its application to the construction of high assurance hardware [1]–[4]. This paper describes the Coq formalization of ReWire intended to support the verification of hardware designs and, in particular, the sort of information flow properties described in our previous work [1], [2], [5].

The aforementioned previous work, in panoramic view, used "by-construction" properties of layered monads to verify properties by hand. For the moment, we rely on the reader's intuition to explain the contributions of the present work at a high level (Section II presents an overview of these concepts in more detail). Assume, for example, that ReWire devices h and l are written respectively in terms of state monad layers, `StT Hi` and `StT Lo`. Then, device h (resp., l) only accesses internal storage of type `Hi` (resp., `Lo`). In a composite device written in terms of monad `M = StT Hi (StT Lo Id)`, it is guaranteed by semantic properties of the layers `StT Hi` and `StT Lo` to disallow covert channels between the `Hi` and `Lo` storage.

The challenge is, then, the formalization of ReWire and, in particular, ReWire's underlying layered monad language and its semantic properties within an automated proof system. The contributions of this work are as follows. **(1)** A static effect-type system (extending Wadler [6]) that disallows covert storage channels in ReWire. This type system extends state layers with effect labels, so that, continuing the example above, h (resp., l) is written in monad `StT RW Hi (StT` $\langle\rangle$ `Lo Id)` (resp., `StT` $\langle\rangle$ `Hi (StT RW Lo Id)`). The effect label "`RW`" means h can both read and write on the `Hi` layer and while "$\langle\rangle$" means it can do neither on the `Lo` layer (and, *vice versa*, for l). The soundness of our type system (Theorems 6 and 7) guarantees freedom from covert storage channels. **(2)** A small-step semantics for ReWire formalized in Coq that justifies **(3)** a typed equational logic (Figure 11) capturing the semantic properties of monads and state layers used in by-hand proofs in our previous work. Finally, **(4)** a number of related metatheorems (e.g., progress, preservation, strong normalization, etc.) have been proved in Coq. All of the definitions and theorems in this paper have been checked with the Coq proof checker (Coq scripts are available here).

The most direct approach to formalizing ReWire in Coq would seem to be the transliteration of monad transformer declarations from Haskell into Coq, but this quickly runs afoul of Coq's strict positivity requirement. ReWire relies on reactive resumption monad transformers (see Section II) as a model of synchronous parallelism and this transformer is a coinductive construction, which can be tricky to formalize, even with Coq's coinduction library. Another approach considers formalizing ReWire's denotational semantics [7], building on existing work by Huffman [8] or Schröder and Mossakowski [9] in Isabelle/HOLCF. Instead, we chose to formalize a small-step, operational semantics for ReWire in Coq, in part, because the authors have more experience with Coq than with HOLCF, but also because developing and formalizing a small-step operational semantics seemed more straightforward than mechanizing denotational semantics. The semantic properties of ReWire's underlying monads on which the by-hand verifications of our previous work rely are then captured as an typed equational logic whose rules are derived from the formalized operational semantics.

The remainder of this section discusses related work. Section II presents an overview of ReWire to motivate the syntax and semantics of the formal calculus, RWC. Section III defines the syntax and small-step operational semantics of RWC. Section IV describes RWC's metatheory and a number of related metatheorems (e.g., progress, preservation, strong normalization, etc.) are demonstrated. A type-directed equational logic for RWC is defined in Section V and Section VI discusses conclusions and future work.

Related Work

Effect systems are a static semantics of effects while monads [10] are a dynamic semantics of effects. Effect systems [11] are commonly associated with impure, strongly-typed functional languages in which the effect annotations make explicit the side effects already present implicitly in the language itself. Monads are used to mimic side-effecting computations within pure, strongly-typed functional languages (e.g., Haskell) in which there are no implicit side effects.

Layered monads—i.e., monads constructed by monad transformers [12]—provide modularity to the semantics of computational effects and functional programs alike by integrating multiple effects within a single monad. This modularity-via-integration, however, has consequences for formal verification: because its effects are all encapsulated within the single monad, they are not distinguished syntactically within the type system of a specification language itself. Wadler [6] "married" effect types to monads, and previous work by the authors [13] seems to be the first marriage of effect types to layered monads. This latter marriage seems to be important for exploiting monadic semantics in formal methods: layered monads provide a modular semantics of effects including by-construction properties and effect types allow the expression of these properties in a formal proof system like Coq (e.g., Figure 11).

As a concept for formal (i.e., machine-checked) verification, monads are less common, although not unheard of [9], [14]–[16] and the use of both effect types and layered monads distinguishes the current work from these.

Formal methods for secure hardware are generally spread across two categories: (1) type-based approaches [17]–[19]; and (2) logic-based approaches (including theorem-proving [20], and BDDs and model-checking [21]), in which a hardware design and desired properties are formulated in a logic and scrutinized in a (semi-)automatic manner. Types-based approaches have support for security concerns integrated into a domain-specific language for hardware description. As with any security type system, however, the question of its permissiveness arises—i.e., does it reject secure designs? The types-based approach offers no recourse to the rejection of a secure design—you simply can't argue with a type checker. A logic-based approach avoids this pitfall, but comes with overhead—e.g., your own theory of security—and, furthermore, it is not connected directly to any implementation path.

One language-based approach to hardware security is to extend an existing HDL with security types. Caisson [17], Sapper [18], and SecVerilog [19] each extend a subset of Verilog with security types and annotations. The type systems of Caisson and SecVerilog reject programs that violate information flow policies, while Sapper uses static analysis to automatically insert dynamic checks to enforce information flow policies at runtime. SecVerilog has an operational semantics, albeit not one formalized in a theorem prover with a proof system [22]. ReWire (or, RWC, rather) differs fundamentally from these language- and type-based approaches in three respects: (1) it is a pure functional language; (2) it possesses a formal semantics mechanized in Coq; and (3) its type system is based on effect types. We discuss the significance of item (3) in Section VI.

The SAFE project focuses on the clean slate design of a provably secure computer system stack (e.g., hardware, operating system, etc.). In a recent publication [23], the SAFE team describes an operational semantics of the SAFE hardware's instruction set and its role in the end-to-end verification in Coq of a non-interference security property. The ReWire project has complimentary, but orthogonal, goals to SAFE: developing a verifiable toolchain for producing high assurance, secure hardware. Interesting follow-on research would explore implementations of the SAFE hardware in the ReWire language.

One traditional approach to hardware verification starts from a design expressed in a production HDL, creates an abstract specification "by hand" as it were, encodes this specification in the logic of an automated theorem prover, and proceeds towards formal verification [20]. This approach relies heavily on the faithfulness of the abstraction step. One reason that this approach must be accomplished "by hand" is that production HDLs do not possess rigorous semantics. Although attempts have been made in the past to define them semantically, none of these projects were evidently completed [24], [25]. By contrast with production HDLs like Verilog or VHDL, ReWire possesses a rigorous semantics for which the present work provides a Coq mechanization. ReWire becomes a vehicle for expressing and implementing hardware designs and for verifying them as well. In previous work [1], [2], we presented several case studies in hardware verification based in ReWire, but there the verifications were not machine-checked.

Goncharov and Schröder [26] extend Moggi's computational λ-calculus with constructs for concurrency and shared state; RWC's design is inspired, in part, by their treatment of corecursion. Crary et al. [27] consider a logical characterization of information flow security that incorporates Moggi's computational λ-calculus at its core. With their approach, monads are, in effect, logical modalities signifying the potential presence of effects at a security level. In contrast, Harrison and Hook's treatment of information flow security [5] is more semantic and model-theoretic than Crary's logical and type-theoretic approach, relying as it does on structural properties of monads and monad transformers to construct secure systems. Security verifications of ReWire designs [1] are based on Harrison and Hook's approach, and the present work formally supports that approach in Coq.

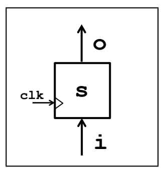
Ghica and Jung [28] provide a categorical semantics for a class of digital circuits in terms of monoidal categories and are motivated by the need for supporting syntactic, equational reasoning. ReWire specifications may be reasoned about equationally in the usual manner of functional languages; this was the approach taken in our previous ReWire verification work [1], [2]. By contrast with Ghica and Jung's work, ReWire specifications are, more or less, ordinary functional programs that are compiled into circuits.

II. Background: ReWire's Programming Model

The purpose of this section is twofold: (1) to make this article as self-contained as possible by providing sufficient background on ReWire and (2) to motivate RWC's type system and operational semantics. Throughout this section, we explicitly link this background material to subsequent sections on RWC. ReWire is a subset of Haskell and uses ideas from monadic semantics as an organizing principle of the language. It is, therefore, assumed of necessity that the reader is, at least, somewhat familiar with functional programming and monads. For those requiring more information about ReWire and its programming model and language design, please consult the references [1], [7].

ReWire is a subset of the Haskell functional programming language [29]—i.e., ReWire programs are Haskell programs, but not necessarily *vice versa*. All ReWire programs can be compiled to synthesizable VHDL with the ReWire compiler. The principal difference between Haskell and ReWire is that recursion in ReWire is restricted to tail recursion so that every ReWire program requires only a finite, bounded memory footprint. Unbounded recursion requires an unbounded stack or heap for compilation and such dynamic control structures are anathema to hardware's fixed storage.

ReWire has type constructors for devices where a *device* represents a clocked computation that, for each clock cycle, takes an input of type `i`, produces an output of type `o`, and may possess internal storage of type `s` (see inset figure). The type of `d` as shown would be `d :: ReT i o (StT s Id) ()`, where `ReT` and `StT` are the reactive resumption and state monad transformers and `Id` is the identity monad (about all of which we say more below in the next sections). Device `d` is clocked, as illustrated in the inset figure, although the clock is represented by the underlying structure of reactive resumptions rather than as an explicit parameter. A device is created in ReWire by either iterating a function or through composition of existing devices. Previous work [4] introduced operators for constructing devices and composing them into larger, interconnected devices; Section II-C presents a simple device specification template in ReWire.



Device d

A. Background: Monads

A monad is a triple $\langle \mathsf{M}, \mathsf{return}, >\!\!>\!\!= \rangle$ consisting of a type constructor M and two operations:

$$\begin{array}{rcll} \mathsf{return} & : & \mathsf{a} \rightarrow \mathsf{M\,a} & \text{— “unit”} \\ (>\!\!>\!\!=) & : & \mathsf{M\,a} \rightarrow (\mathsf{a} \rightarrow \mathsf{M\,b}) \rightarrow \mathsf{M\,b} & \text{— “bind”} \end{array}$$

These operations must obey the well-known *monad laws* [10], [12] (these are (LEFT-UNIT), (RIGHT-UNIT), and (ASSOCIATIVITY) in Figure 11). The return operator is the monadic analogue of the identity function, injecting a value into the monad. The $>\!\!>\!\!=$ operator is a form of sequential application. The “null bind” operator, $>\!\!> \,:\, \mathsf{M\,a} \rightarrow \mathsf{M\,b} \rightarrow \mathsf{M\,b}$, is defined as: $\mathsf{x} >\!\!> \mathsf{k} \;=\; \mathsf{x} \;>\!\!>\!\!=\; \lambda\_.\mathsf{k}$. The binding (i.e., “$\lambda\_$”) acts as a dummy variable, ignoring the value produced by x.

B. Background: Monad Transformers

The organizing principle underlying ReWire are *reactive resumption monads with state* [30] (RRMS), which encapsulate a notion of computation appropriate to hardware—namely, synchronous parallelism. RRMS support the expression of structural hardware designs in a functional style [4]. RWC is a computational λ-calculus whose syntax and semantics formalizes RRMS in Coq. In particular, RWC's type system includes constructors that correspond to the state and reactive resumption monad transformers. For the sake of being self-contained, we provide the reader with Haskell definitions of the `StT` and `ReT` monad transformers. This code is meant only to aid the reader in comprehending the intended semantics of RWC. If more background is required on RRMS, please consult the references [30].

1) State Monad Transformer: The state monad transformer is a well-documented structure in functional programming and semantics [12]. The Haskell code for the state monad transformer, `StT`, along with its `lifting` functions is below:

```
data StT s m a = StT (s -> m (a,s))
lift_StT :: m a -> StT s m a
lift_StT m
     = StT (\ s -> m >>=_m \ v -> return_m (v,s))
get :: StT s m s
get = StT (\ s -> return_m (s,s))
put :: s -> StT s m ()
put s = StT (\ _ -> return_m ((),s))
```

The `lift` converts or “lifts” an `m a` computation into an `StT s m a` computation. The `get` operation returns the current value of the `s`-store while the `put s` operation replaces the current store with store `s`. In the definitions above, the binds and returns for the `m` monad are affixed with a subscript to disambiguate them from the operations being defined.

2) Reactive Resumption Monad Transformer: Computations in `ReT i o m a` may be viewed intuitively as (potentially infinite) sequences of `m` computations. If that sequence terminates, it produces an `a`-value, otherwise it produces an `o`-output value and a continuation. Both `lift` operations convert an `m` computation into respective enriched computations. Computations in `ReT` over layered state monads correspond closely to synchronous hardware as discussed in previous work [1].

The Haskell code for the reactive resumption monad transformer, `ReT`, along with its associated functions is below:

```
data ReT i o m a
  = Pause (m (Either a (o,i -> ReT i o m a)))
lift_ReT :: m a -> ReT i o m a
lift_ReT m = Pause (m >>=_m return_m . Left)
signal :: o -> ReT i o m i
signal o = Pause (return_m
                 (Right (o,return_m . return_ReT)))
data Either a b = Left a | Right b
```

Recall that function composition (i.e., “`.`”) and sum types (i.e., `Either`) are built-in to Haskell. In terms of the device `d` example above, the operation `signal o` represents the end of a clock cycle and sets the output signal of `d` to `o`. RWC includes a `pause` primitive in the term syntax (Fig. 2) as a means of representing `signal`.

3) By-construction Properties of Layered Monads: Layered state monads—monads with multiple `StT` applications (e.g., $\mathtt{M} = \mathtt{StT}\,\mathtt{s}_1\,(\mathtt{StT}\,\mathtt{s}_2\,\mathtt{Id})$)—have a number of useful properties by construction [5], including:

$$\mathtt{put}\ \mathsf{s}' \gg \mathtt{put}\ \mathsf{s} \quad = \mathtt{put}\ \mathsf{s}$$
$$\mathtt{put}\ \mathsf{s} \gg \mathtt{lift}_{\mathtt{StT}}\ \varphi = \mathtt{lift}_{\mathtt{StT}}\ \varphi \gg \mathtt{put}\ \mathsf{s}$$

The first rule is an inter-layer property (a.k.a., "clobber") while the second is an inter-layer property (a.k.a., "atomic non-interference"). Clobber states that the put s cancels earlier effects on the same layer. By convention for a fixed state s_0, we define mask $=$ put s_0; the mask included in the term syntax of RWC generalizes this idea. Atomic non-interference states that effects from different state layers commute. The equational logic derived in Coq for RWC presented in Section V gives generalizations of both properties.

C. Defining Devices in ReWire

Simple ReWire devices are generally defined as tail recursive functions whose codomain is written in terms of the ReT layer. Assume that we have functions defined which specify the internal and external behaviors of device d that have function types: `internal :: i -> StT s Id v` and `external :: i -> v -> o`. Function internal takes the input i, performs some computation with the current internal storage s, and produces an intermediate result v. Function external takes the input i and the result v and produces the next output signal for d.

Given an initial input $\mathtt{i}_0$, d $=$ dev $\mathtt{i}_0$ where corecursive function dev is defined as:

```
dev :: i -> ReT i o (StT s Id) ()
dev i = liftReT (internal i)   >>= \ v ->
        signal (external i v) >>= \ i' ->
        dev i'
```

At the beginning of a clock cycle, dev first consumes input, i, then performs internal i computation on the internal storage s, and then outputs the external i v signal at the end of the clock cycle.

Device definitions are expressed with an explicit corecursion operator, unfold; for example, device d would be written:

```
unfold i0
     (\ i -> internal i >>= \ v ->
             return (Right (external i v, id)))
```

For this reason, Figure 2 includes syntax for an unfold primitive and its semantics are defined in subsequent sections.

D. Background: Goguen-Meseguer Non-interference

The essence of the Goguen-Meseguer noninterference information flow model [31] and its many descendants is that systems, broadly construed, are state machines whose inputs and outputs are partitioned by security level. The definition of information flow is formulated in terms of sequences of stateful operations of mixed security levels and stipulates that high-level operations must not affect low-level outputs. More concretely, for any mixed-level sequence, $\mathtt{s} = (\mathtt{l}_1\,;\,\mathtt{h}_1\,;\,\ldots\,;\,\mathtt{l}_n\,;\,\mathtt{h}_n)$, the low-level outputs of s must be identical to those produced by $(\mathtt{l}_1\,;\,\ldots\,;\,\mathtt{l}_n)$, which is the result of filtering out from s all high-level operations.

E. The Marriage of Effects and Layered State Monads

"By construction" properties of layered state monads [5] tell us that high- and low-security operations commute (a.k.a., atomic non-interference) and that mask_H cancels high-level operations (i.e., $\varphi_H \gg \mathsf{mask}_H = \mathsf{mask}_H$). This cancelling property is known as the "clobber rule" [5]. The atomic non-interference and clobber rules are helpful in demonstrating that monadic noninterference equations (like that of the previous section) hold for particular software and hardware applications [1], [5].

The Goguen-Meseguer model was recast in monadic terms previously [5], so that high-level effects must be cancellable without affecting the low-level effects. Here, the utility of the RWC effect type system becomes evident, because it can statically distinguish computations occurring on distinct layers. For the sake of concreteness, consider the case of a monad, M, with a high- and low-security stores types, H and L. High and low operations may be distinguished by the RWC effect type system by annotating the layers with effect labels:

$$\varphi_H : \mathsf{StT\ RW}\ H\ (\mathsf{StT}\ \langle\rangle\ L\ \mathsf{Id})()$$
$$\varphi_L : \mathsf{StT}\ \langle\rangle\ H\ (\mathsf{StT\ RW}\ L\ \mathsf{Id})()$$

Note that φ_H (resp., φ_L) only has read-write effects (RW) on the outer (resp., inner) state layer of M. Furthermore, we assume the existence of an operation, mask_H which initializes the H state layer. The mask_H operation can be assumed to be put s_0 on the H-layer, where s_0 is an arbitrary, fixed value in H. Then, the monadic formulation of non-interference boils down to demonstrating that equations like the following hold:

$$\varphi_H \gg \varphi_L \gg \mathsf{mask}_H = \varphi_L \gg \mathsf{mask}_H$$

Put simply, this means that reinitializing the H layer cancels the effects of high-security operations like φ_H. This is the monadic analogy of Goguen and Meseguer's filtering out of high-security operations.

III. RWC: The ReWire Core Calculus

This section introduces the syntax (Section III-A), type system (Section III-B) and operational semantics (Section III-C) of the ReWire Calculus (RWC). RWC is a computational λ-calculus that extends the functional features of a typed lambda calculus with support for stateful effects and reactive parallelism. These effects are encapsulated through the use of *monads* [10], enabling us to provide a useful equational theory in the presence of effects. The addition of effects to a computational λ-calculus was examined in [6].

A. Syntax

This section introduces the syntax of RWC, which is a variety of computational λ-calculus extended with operations for synchronous, stateful parallelism. Here, the stateful component is organized as *layered* state monads—i.e., monads created by multiple applications of the state monad transformer. Layered state monads have by-construction properties that support information flow security verification [1], [5]; we defer presenting the general formalization of these by-construction

$$
\begin{aligned}
\ell \in \mathit{EffectLabel} &::= \langle\rangle \mid \mathsf{R} \mid \mathsf{W} \mid \mathsf{RW} \\
\mathsf{S} \in \mathit{StateMonad} &::= \mathsf{Id} \mid \mathsf{StT}\ \ell\ \tau\ \mathsf{S} \\
\mathsf{M} \in \mathit{Monad} &::= \mathsf{S} \mid \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S} \\
\tau, \tau' \in \mathit{Type} &::= \tau \to \tau' \mid \tau \times \tau' \mid \tau + \tau' \mid () \mid \mathsf{M}\ \tau
\end{aligned}
$$

Figure 1: Syntax of RWC types

$$
\begin{aligned}
\mathit{Identifier} &::= x \mid y \mid z \mid w \mid \mathit{etc.} \\
t \in \mathit{Term} &::= x \mid t\ t' \mid \lambda x : \tau . t \mid () \mid \langle t, t' \rangle \mid \mathsf{proj}\ t\ t' \mid \\
&\quad \mid \mathsf{inl}_\tau\ \ t \mid \mathsf{inr}_\tau\ \ t \mid \mathsf{case}\ t\ t'\ t'' \mid \mathsf{return}_\mathsf{M}\ \ t \mid t >\!\!>\!= t' \\
&\quad \mid \mathsf{lift}_\mathsf{M}\ \ t \mid \mathsf{elevate}_\mathsf{S}\ t \mid \mathsf{get}_\mathsf{S} \mid \mathsf{put}\ \ t \mid \mathsf{pause}_{\mathsf{M},\tau}\ \ t \\
&\quad \mid \mathsf{runSt}\ \ t\ t' \mid \mathsf{runId}\ t \mid \mathsf{unfold}_{\mathsf{M},\tau,\tau'}\ \ t\ t' \mid \mathsf{runRe}_\tau\ \ t \\
v, s \in \mathit{Value} &::= \lambda x : \tau . t \mid () \mid \langle v, v' \rangle \mid \mathsf{inl}_\tau\ \ v \mid \mathsf{inr}_\tau\ \ v \mid \mathsf{return}_\mathsf{M}\ \ v \\
&\quad \mid v >\!\!>\!= v' \mid \mathsf{lift}_\mathsf{M}\ \ v \mid \mathsf{elevate}_\mathsf{M}\ \ v \mid \mathsf{get}_\mathsf{S} \mid \mathsf{put}\ \ v \\
&\quad \mid \mathsf{pause}_{\mathsf{M},\tau}\ \ v \mid \mathsf{runSt}\ \ v\ v' \mid \mathsf{runRe}_\tau\ \ v \\
&\quad \mid \mathsf{unfold}_{\mathsf{M},\tau,\tau'}\ \ v\ v' \\
\Sigma \in \mathit{Store} &::= \mathsf{nil} \mid s :: \Sigma \\
c \in \mathit{Config} &::= \langle t, \Sigma \rangle \\
D \in \mathit{DoneConfig} &::= \langle\, \mathsf{return}_\mathsf{M}\ \ v, \Sigma \rangle \mid \langle\, \mathsf{pause}_{\mathsf{M},\tau}\ \ v, \Sigma \rangle
\end{aligned}
$$

Figure 2: Syntax of terms, stores, and configurations

properties until Section V. Section II provides the reader with some background on monad transformers, although readers requiring more should consult the references.

1) Types: Figure 1 shows the syntax of types.

As a computational λ-calculus, RWC extends the simply-typed λ-calculus with unit, sum, and product types along with a notion of *computational* types: if M is a monad and τ is a type, then $M\ \tau$ is the type of computations in the monad M with a result value of type τ. Exactly which monad stands in for M will determine what sort of computational effects are possible. RWC permits the use of monads built in terms of the Id (identity) monad and the ReT (reactive resumption), and StT (state) monad transformers, where ReT must be the outermost monad transformer application (if it is present). RWC's monads encompass the combination of resumption and layered state monads found in [30] with the addition of *effect labels* attached to each StT. The presence of an effect label ℓ at a given layer certifies that the computation has at *most* the effects ℓ at that layer. For example, the effect label W reflects the *possibility* that a computation will write, not the necessity, and certifies that the computation will *not* read.

We note in passing that the *denotational* semantics of these monads corresponds exactly to the semantics of their Haskell equivalents, up to the erasure of the effect labels and with the considerable simplification that lifted domains are not necessary due to the absence of general recursion; see [7] for further details.

2) Terms: Figure 2 shows the syntax of terms. Note the widespread use of type and monad subscripts. These are necessary to ensure that every term has a unique type, and to handle overloading of monadic operations. We will sometimes omit these subscripts, as long as doing so does not introduce ambiguity.

We will not remark on the standard λ-calculus machinery, other than to note that the constructs used for destructing pairs and elements of sum type are slightly nonstandard. The term constructor proj, used for destructing pairs, takes two subterms: the first corresponding to the pair being deconstructed—suppose it has type $\tau \times \tau'$—and the second corresponding to a *function* of type $\tau \to \tau' \to \tau''$ that produces a value from the pair's elements. (Note that the conventional left- and right-projection operators can be constructed in terms of the proj operator.) The term constructor case, used for destructing elements of sum type, takes three subterms: the first is the scrutinee of type $\tau + \tau'$, the second to a function f_1 of type $\tau \to \tau''$, and the third to a function f_2 of type $\tau' \to \tau''$. If the scrutinee evaluates to inl v (resp., inr v), then v will be passed to f_1 (resp., f_2).

Computations are defined in terms of certain primitives. The (overloaded) term constructors return and $\star$ correspond respectively to the unit and bind operations of the monads, and lift to the lift operation of each monad transformer. Terms typed in a state monad may read and write to the store using the get and put operations. The term constructor elevate adds effect labels – e.g., W or R – to the effect labels, if any, on a state monad computation; thereby, converting state monad computations with a less permissive types to a more permissive type (where "permissiveness" is understood as in Figure 4). For example, a term t of type StT R τ Id τ' can be typecast into the more permissive type StT RW τ Id τ' via elevate, essentially de-certifying that t does not write. (A cast in the "other direction", to StT $\langle\rangle$ τ Id τ', is not permitted by the type system.) Reactive computations are defined in terms of the primitives pause and unfold. The term pause t is essentially a suspended computation that is waiting for an input value, and unfold can be used to produce "looping" computations; we postpone a discussion of their exact semantics until we have discussed the type system in greater detail. Finally, the term constructors runRe, runSt, and runId allow the effects of a given monad transformer to be reflected into the base monad. It may be helpful to view runRe as executing a single step of a resumption-monadic computation, runSt as supplying the initial state for the uppermost state layer, and runId as moving from the effect-free Id monad into the universe of non-monadic terms.

3) Stores and Configurations.: Figure 2 (bottom) shows the syntax of *stores* and *configurations*, which will be used to specify the semantics of computations. A store is a list of terms, each of which corresponds semantically to a state monad transformer, and a configuration $\langle t, \Sigma \rangle$ pairs a term t with a state Σ. Generally, we use the metavariables s, s', s'' to refer store values.

B. Type System

Typing rules for terms are given in Figure 3. Typing judgments take the form $\Gamma \vdash t : \tau$, where Γ is a set of assumptions (i.e., a mapping of variables to types). For the

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{return}_{\mathsf{M}}\, t : \mathsf{M}\ \tau}\ (\textsc{Return}) \qquad \frac{\Gamma \vdash t : \mathsf{M}\ \tau \quad \Gamma \vdash t' : \tau \to \mathsf{M}\ \tau'}{\Gamma \vdash t \mathrel{>\!\!>\!\!=} t' : \mathsf{M}\ \tau'}\ (\textsc{Bind})$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ \tau}{\Gamma \vdash \mathsf{lift}_{(\mathsf{StT}\ell\tau'\mathsf{S})}\ t : \mathsf{StT}\ \ell\ \tau'\ \mathsf{S}\ \tau}\ (\textsc{LiftSt})$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ \tau}{\Gamma \vdash \mathsf{lift}_{(\mathsf{ReT}\tau'\tau''\mathsf{S})}\ t : \mathsf{ReT}\ \tau'\ \tau''\ \mathsf{S}\ \tau}\ (\textsc{LiftRe})$$

$$\frac{\mathsf{R} \leq \ell}{\Gamma \vdash \mathsf{get}_{(\mathsf{StT}\ell\tau\mathsf{S})} : \mathsf{StT}\ \ell\ \tau\ \mathsf{S}\ \tau}\ (\textsc{get}) \qquad \frac{\Gamma \vdash t : \tau \quad \mathsf{W} \leq \ell}{\Gamma \vdash \mathsf{put}\ \ t : \mathsf{StT}\ \ell\ \tau\ \mathsf{S}\ ()}\ (\textsc{put})$$

$$\frac{\Gamma \vdash t : \mathsf{StT}\ \ell\ \tau'\ \mathsf{S}\ \tau \quad \Gamma \vdash t' : \tau'}{\Gamma \vdash \mathsf{runSt}\ \ t\ t' : \mathsf{S}\ (\tau \times \tau')}\ (\textsc{RunSt}) \qquad \frac{\Gamma \vdash t : \mathsf{Id}\ \tau}{\Gamma \vdash \mathsf{runId}\ t : \tau}\ (\textsc{RunId})$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ (\tau' \times (\tau \to \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau''))}{\Gamma \vdash \mathsf{pause}_{(\mathsf{ReT}\tau\tau'\mathsf{S},\tau'')}\ \ t : \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau''}\ (\textsc{pause})$$

$$\frac{\Gamma \vdash t : \tau''' \quad \Gamma \vdash t' : \tau''' \to \mathsf{S}\ (\tau'' + (\tau' \times (\tau \to \tau''')))}{\Gamma \vdash \mathsf{unfold}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S},\tau'',\tau''')}\ \ t\ t' : \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau''}\ (\textsc{unfold})$$

$$\frac{\Gamma \vdash t : \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau''}{\Gamma \vdash \mathsf{runRe}_{\tau}\ \ t : \mathsf{S}\ (\tau'' + (\tau' \times (\tau \to \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau'')))}\ (\textsc{RunRe})$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ \tau \quad \mathsf{S} \leq \mathsf{S}'}{\Gamma \vdash \mathsf{elevate}_{\mathsf{S}'}\ \ t : \mathsf{S}'\ \tau}\ (\textsc{Elevate})$$

Figure 3: Typing judgments for terms. Rules for λ-calculus are omitted.

$$\begin{tikzcd}[column sep=small] & \mathsf{RW} \arrow[dl, dash] \arrow[dr, dash] & \\ \mathsf{R} \arrow[dr, dash] & & \mathsf{W} \arrow[dl, dash] \\ & \langle\rangle & \end{tikzcd} \qquad \frac{}{\mathsf{Id} \leq \mathsf{Id}}\ (\text{L-Id}) \qquad \frac{\ell \leq \ell' \quad S \leq S'}{\mathsf{StT}\ \ell\ \tau\ S \leq \mathsf{StT}\ \ell'\ \tau\ S'}\ (\text{L-StT})$$

Figure 4: Ordering on effect labels (given by the diagram) and on state monads.

empty context, we write $\{\}$. Many of the rules are standard, reflecting the rules of computational λ-calculus. The rules for get, put, and elevate require special attention, as they directly involve effect labels. Rule T-GET restricts the effect label on the top monad transformer to include a read label, and T-PUT restricts it to include a write label. These restrictions are expressed in terms of an ordering on effect labels (which is really nothing more than the subset relation) given in Figure 4 at left. For rule T-ELEVATE, we require that the target monad S' has (non-strictly) more effect labels than the source monad S; the precise meaning of this is expressed in Figure 4 at right. The intuition is that elevate permits us to *decertify* that a computation does *not* read or write at any given state layers, but not to remove existing effect labels.

Stores and configurations also have a notion of type, defined by the rules of Figure 5. A store Σ is said to *match* a monad M if the types of its elements correspond, in order, to the state types of the state monad transformers in M. For this, we simply write that Σ matches M. A configuration $\langle t, \Sigma \rangle$, then, has type $M\ \tau$ if and only if Σ matches M and $\{\} \vdash t : M\ \tau$. We write this $\langle t, \Sigma \rangle \triangleright M\ \tau$.

Theorem 1 (Uniqueness of Types): If $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \tau'$, then $\tau = \tau'$. Also, if $\langle t, \Sigma \rangle \triangleright \tau$ and $\langle t, \Sigma \rangle \triangleright \tau'$, then $\tau = \tau'$.

$$\frac{\Sigma \text{ matches } \mathsf{S}}{\Sigma \text{ matches } \mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}}\ (\textsc{M-Ret}) \qquad \frac{\{\} \vdash s : \tau \quad \Sigma \text{ matches } \mathsf{S}}{s :: \Sigma \text{ matches } \mathsf{StT}\ \ell\ \tau\ \mathsf{S}}\ (\textsc{M-StT})$$

$$\frac{\{\} \vdash t : \mathsf{M}\ \tau \quad \Sigma \text{ matches } \mathsf{M}}{\langle t, \Sigma \rangle \triangleright \mathsf{M}\ \tau}\ (\textsc{T-Config})$$

Figure 5: Typing judgments for stores (top) and configurations (bottom).

C. Small-Step Operational Semantics

In this section we describe the semantics of RWC in a small-step operational style. As a computational λ-calculus, RWC contains both functional features (functional abstraction and application) as well as effectful ones (mutable state and reactive parallelism). The operational semantics is structured around this dichotomy, with two interdefined notions of reduction: *pure* and *effectful* reduction. Pure reduction reflects the notion of effect-free evaluation. A pure reduction judgment takes the form $t \rightsquigarrow t'$; note that this makes no mention of any store. Effectful reduction provides semantics to computational terms which may have effects. Thus an effectful reduction judgment takes the form $\langle t, \Sigma \rangle \rightsquigarrow \langle t', \Sigma' \rangle$.

The rules for pure and effectful reduction are given in Figures 6 and 7, respectively. We adopt a call-by-value evaluation strategy, as this is (we feel) simpler to work with metatheoretically than call-by-name or -need. This may seem strange in light of ReWire's antecedents in Haskell (which is a non-strict language), but since ReWire is a strongly normalizing subset of Haskell, it does not matter whether we choose an eager or lazy evaluation strategy from a "backwards compatibility" point of view: since there is no "bottom" value, strictness is not a concern.

A few of the rules require close inspection. To begin with, we note that pure and effectful reduction are interdefined. Rule STM-ST of Figure 7 allows pure reduction to be "lifted" into the universe of effectful reduction: if the term component t of a configuration $\langle t, \Sigma \rangle$ still has not been evaluated to a value, we will continue to evaluate it without changing the store. Dually, if less obviously, the rule ST-RUNIDMO in Figure 6 allows monadic evaluation in the identity monad (and *only* in the identity monad) to be reified in a pure setting. If we wish to run a computation in a more complex monad, we may use runRe and runSt to "peel off" one monad transformer at a time, until we reach the Id monad at the core. In the runSt case, we must supply an initial value for the corresponding state layer, producing a computation in the base monad which will return the post-value for that layer. The runRe operator will produce a computation in the base monad that either returns a final result value, or an output value paired with a continuation waiting on more input.

Note also the interaction between the rule STM-LIFTST, STM-GET, and STM-PUT. The get and put operations always operate on the 'head' (leftmost) item in the store. Applying $\mathsf{lift}_{\mathsf{StT}}$ to these operations allows us to access items deeper in the store, by executing the underlying computation against the "tail" of the store and leaving the "head" item unchanged.

$$\frac{}{(\lambda x : \tau . t)v \rightsquigarrow t[x := v]}\ (\textsc{ST-AppAbs}) \qquad \frac{t \rightsquigarrow t''}{t\ t' \rightsquigarrow t''\ t'}\ (\textsc{ST-App1})$$

$$\frac{t \rightsquigarrow t'}{v\ t \rightsquigarrow v\ t'}\ (\textsc{ST-App2}) \qquad \frac{t \rightsquigarrow t'}{\mathsf{inl}_\tau\ \ t \rightsquigarrow \mathsf{inl}_\tau\ \ t'}\ (\textsc{ST-Inl}) \qquad \frac{\langle v, \mathsf{nil}\rangle \rightsquigarrow \langle t, \mathsf{nil}\rangle}{\mathsf{runId}\ v \rightsquigarrow \mathsf{runId}\ t}\ (\textsc{ST-RunIdMo})$$

Figure 6: Reduction Rules for Lambda Calculus Reduction. These rules, mostly omitted, specify a call-by-value evaluation strategy on RWC.

$$\frac{}{\mathsf{nil} \stackrel{\mathsf{Id}(\text{₩})}{=} \mathsf{nil}}\ (\textsc{swnw-id}) \qquad \frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{\Sigma \stackrel{\mathsf{ReT}\ \tau\ \tau'\ S(\text{₩})}{=} \Sigma'}\ (\textsc{swmw-re})$$

$$\frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{(s :: \Sigma) \stackrel{\mathsf{StT}\ \ \tau \langle\rangle\ S(\text{₩})}{=} (s :: \Sigma')}\ (\textsc{swnw-n}) \qquad \frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{(s :: \Sigma) \stackrel{\mathsf{StT}\ \ \tau \langle R\rangle\ S(\text{₩})}{=} (s :: \Sigma')}\ (\textsc{swnw-r})$$

$$\frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{(s :: \Sigma) \stackrel{\mathsf{StT}\ \ \tau \langle W\rangle\ S(\text{₩})}{=} (s' :: \Sigma')}\ (\textsc{swnw-w})$$

$$\frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{(s :: \Sigma) \stackrel{\mathsf{StT}\ \ \tau \langle RW\rangle\ S(\text{₩})}{=} (s' :: \Sigma')}\ (\textsc{swnw-rw})$$

Figure 8: The 'same where no write' relation.

$$\frac{}{\mathsf{nil} \stackrel{\mathsf{Id}(\text{₩})}{=} \mathsf{nil}}\ (\textsc{swr-id}) \qquad \frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{\Sigma \stackrel{\mathsf{ReT}\ \tau\ \tau'\ S(\text{₩})}{=} \Sigma'}\ (\textsc{swr-re})$$

$$\frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{(s :: \Sigma) \stackrel{\mathsf{StT}\ \ \tau \langle\rangle\ S(\text{₩})}{=} (s' :: \Sigma')}\ (\textsc{swr-n}) \qquad \frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{(s :: \Sigma) \stackrel{\mathsf{StT}\ \ \tau \langle R\rangle\ S(\text{₩})}{=} (s :: \Sigma')}\ (\textsc{swr-r})$$

$$\frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{(s :: \Sigma) \stackrel{\mathsf{StT}\ \ \tau \langle W\rangle\ S(\text{₩})}{=} (s' :: \Sigma')}\ (\textsc{swr-w})$$

$$\frac{\Sigma \stackrel{S(\text{₩})}{=} \Sigma'}{(s :: \Sigma) \stackrel{\mathsf{StT}\ \ \tau \langle RW\rangle\ S(\text{₩})}{=} (s :: \Sigma')}\ (\textsc{swr-rw})$$

Figure 9: The 'same where read' relation.

The rule STM-UNFOLD may be justified directly by the Haskell definition of unfold. Rule STM-PAUSE is more subtle. The basic idea, however, is that if a pause arises to the left of a $\star$, we should "absorb" what comes to the right of the $\star$ into the pause's continuation, guaranteeing that we make progress towards a "done" configuration.

As stated in Theorem 2, the reduction relation the results from the rules for pure and effectful reduction is deterministic.

Theorem 2 (Evaluation is Deterministic): If $t \rightsquigarrow t'$ and $t \rightsquigarrow t''$, then $t' = t''$. Also, if $\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma'\rangle$ and $\langle t, \Sigma\rangle \rightsquigarrow \langle t'', \Sigma''\rangle$, then $\langle t', \Sigma'\rangle = \langle t'', \Sigma''\rangle$.

IV. METATHEORY

In this section we discuss the metatheory of RWC, in particular *type safety* (Section IV-A), *strong normalization* (Section IV-B), and *soundness* of effect labels (Section IV-C).

A. Type Safety

As is standard in operational semantics, we take type safety to be the conjunction of *progress*, meaning that any well-typed term (resp. configuration) that is not a value (resp. is not "done") always reduces to something, and *preservation*, meaning that reduction preserves the types of terms (and configurations). Together, these properties imply that well-typed programs can't go wrong—i.e., evaluation of well-typed programs never "gets stuck".

Theorem 3 (Progress): If $\{\} \vdash t : \tau$, then either t is a value or there exists t' such that $t \rightsquigarrow t'$. Also, if $\langle t, \Sigma\rangle \triangleright M\ \tau$, then either $\langle t, \Sigma\rangle$ is done, or there exist t' and Σ' such that $\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma'\rangle$.

Theorem 4 (Preservation): If $\{\} \vdash t : \tau$ and $t \rightsquigarrow t'$, then $\{\} \vdash t' : \tau$. Also, if $\langle t, \Sigma\rangle \triangleright M\ \tau$ and $\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma'\rangle$, then $\langle t', \Sigma'\rangle \triangleright M\ \tau$.

Perhaps surprisingly, the addition of computational features does not substantially complicate the proof of type safety relative to a pure λ-calculus.

B. Normalization

Unlike Haskell, RWC enjoys the property of *strong normalization*, which means that all well-typed terms (resp. configurations) will eventually reduce to some value (resp. done configuration). This property is especially important in hardware applications for the reason that hardware cannot be allowed to "loop forever" between clock ticks.The computation time between clock ticks must have a static, finite upper bound—this issue is discussed in detail in the references [1], [7]. Strong normalization also makes defined equality easier to work with, as it eliminates the need to account for equality of diverging computations.

We shall write $\rightsquigarrow^*$ for the multistep reduction relation, i.e. the reflexive, transitive closure of $\rightsquigarrow$. We say that a term t *halts* if and only if there exists a (not necessarily distinct) value v, such that $t \rightsquigarrow^* v$. In a similar fashion, a configuration $\langle t, \Sigma\rangle$ *halts* if and only if there exists a done configuration D such that $\langle t, \Sigma\rangle \rightsquigarrow^* D$.

Theorem 5 (Normalization): If $\{\} \vdash t : \tau$, then t halts. Also, if $\langle t, \Sigma\rangle \triangleright M\ \tau$, then $\langle t, \Sigma\rangle$ halts.

The proof of Theorem 5 uses an adaptation of the standard *logical relations* technique [32]. Given a property $\mathbf{P}$, a *logical relation*, $\mathcal{R}_{\{\mathbf{T} \in \mathcal{T}\}}$ (with respect to $\mathbf{P}$), is a collection of type-indexed relations such that for every $\mathbf{R_T} \in \mathcal{R}_{\{\mathbf{T} \in \mathcal{T}\}}$, every element $t \in \mathbf{R_T}$, either has, or preserves $\mathbf{P}$. In the case of strong normalization, halting is the property of interest.

Proving Theorem 5 in Coq required developing novel techniques. Because resumptions involve potentially infinite computations, proving that strong normalization holds for configurations requires the use of proofs by coinduction. The use of coinduction allows the $\mathbf{R}$ property to be appropriately applied over potentially infinite computations. The use of coinduction and the corresponding notion of bisimulation have been attributed to David Park [33].

C. Soundness of Effect Labels

Since effect labels are meant to track effects and their potential propagation, soundness of effect labels (roughly) corresponds to preservation of security levels indicated by the label, and that stores track such features accordingly. Thus, given well-typed configurations, establishing soundness of effect labels amounts to verifying that monadic-reduction

$$\frac{t \rightsquigarrow t'}{\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma\rangle}\ \text{(STM-ST)} \qquad \frac{\langle v, \Sigma\rangle \rightsquigarrow \langle t, \Sigma'\rangle}{\langle v \ \texttt{>>=}\ v', \Sigma\rangle \rightsquigarrow \langle t \ \texttt{>>=}\ v', \Sigma'\rangle}\ \text{(STM-BIND)} \qquad \frac{}{\langle \mathsf{return}_{\mathsf{M}}\ v \ \texttt{>>=}\ v', \Sigma\rangle \rightsquigarrow \langle v'\ v, \Sigma\rangle}\ \text{(STM-BINDRET)}$$

$$\frac{\langle v, \Sigma\rangle \rightsquigarrow \langle t, \Sigma'\rangle}{\langle \mathsf{lift}_{(\mathsf{StT}\ \ell\ \tau\ \mathsf{S})}\ v, s :: \Sigma\rangle \rightsquigarrow \langle \mathsf{lift}_{(\mathsf{StT}\ \ell\ \tau\ \mathsf{S})}\ t, s :: \Sigma'\rangle}\ \text{(STM-LIFTST)} \qquad \frac{\langle v, \Sigma\rangle \rightsquigarrow \langle t, \Sigma'\rangle}{\langle \mathsf{lift}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ v, \Sigma\rangle \rightsquigarrow \langle \mathsf{lift}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ t, \Sigma'\rangle}\ \text{(STM-LIFTRE)}$$

$$\frac{}{\langle \mathsf{lift}_{\mathsf{M}}\ (\mathsf{return}_{\mathsf{M}'}\ v), \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{M}}\ v, \Sigma\rangle}\ \text{(STM-LIFTRET)} \qquad \frac{}{\langle \mathsf{get}_{\mathsf{S}}, s :: \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{S}}\ s, s :: \Sigma\rangle}\ \text{(STM-GET)} \qquad \frac{}{\langle \mathsf{put}\ v, s :: \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{S}}\ (), v :: \Sigma\rangle}\ \text{(STM-PUT)}$$

$$\frac{\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma'\rangle}{\langle \mathsf{elevate}_{\mathsf{S}}\ t, \Sigma\rangle \rightsquigarrow \langle \mathsf{elevate}_{\mathsf{S}}\ t', \Sigma'\rangle}\ \text{(STM-ELEVATE)} \qquad \frac{}{\langle \mathsf{elevate}_{\mathsf{S}'}\ (\mathsf{return}_{\mathsf{S}}\ v), \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{S}'}\ v, \Sigma\rangle}\ \text{(STM-ELEVATERET)}$$

$$\frac{\langle v, s :: \Sigma\rangle \rightsquigarrow \langle t, s' :: \Sigma'\rangle}{\langle \mathsf{runSt}\ v\ s, \Sigma\rangle \rightsquigarrow \langle \mathsf{runSt}\ t\ s', \Sigma'\rangle}\ \text{(STM-RUNST)} \qquad \frac{}{\langle \mathsf{runSt}\ (\mathsf{return}_{(\mathsf{StT}\ \ell\ \tau\ \mathsf{S})}\ v)\ v', \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{S}}\ \langle v, v'\rangle, \Sigma\rangle}\ \text{(STM-RUNSTRET)}$$

$$\frac{\langle v, \Sigma\rangle \rightsquigarrow \langle t, \Sigma'\rangle}{\langle \mathsf{runRe}_{\tau}\ v, \Sigma\rangle \rightsquigarrow \langle \mathsf{runRe}_{\tau}\ t, \Sigma'\rangle}\ \text{(STM-RUNRE)} \qquad \frac{}{\langle \mathsf{runRe}_{\tau''}\ (\mathsf{pause}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ v), \Sigma\rangle \rightsquigarrow \langle v \star \lambda \mathsf{x}.\mathsf{return}\ (\mathsf{inr}_{\tau''}\ \mathsf{x}), \Sigma\rangle}\ \text{(STM-RUNREPAUSE)}$$

$$\frac{}{\langle \mathsf{runRe}_{\tau''}\ (\mathsf{return}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ v), \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{S}\ (\mathsf{inl}_{(\tau \to (\tau' \times (\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S}\ \tau'')))}\ v), \Sigma\rangle}\ \text{(STM-RUNRERET)}$$

$$\frac{}{\langle \mathsf{unfold}\ v\ v', \Sigma\rangle \rightsquigarrow \left\langle \mathsf{lift}\ (v'\ v)\ \texttt{>>=}\ \lambda u. \left(\begin{array}{ll}\mathsf{case}\ u & (\lambda w.\ \mathsf{return}\ w) \\ & (\lambda w.\mathsf{proj}\ w\ (\lambda x.\lambda y.\ \mathsf{pause}(\mathsf{return}\langle x, \lambda z.\ \mathsf{unfold}\ (y\ z)\ v'\rangle)))\end{array}\right), \Sigma\right\rangle}\ \text{(STM-UNFOLD)}$$

$$\frac{}{\langle (\mathsf{pause}\ v) \star v', \Sigma\rangle \rightsquigarrow \left\langle \mathsf{pause}\ \left(v\ \texttt{>>=}\ \lambda w.\ \left(\mathsf{proj}\ w\ (\lambda x.\lambda y.\ \mathsf{return}\langle x, \lambda z.(y\ z)\ \texttt{>>=}\ v'\rangle)\right)\right), \Sigma\right\rangle}\ \text{(STM-PAUSEBIND)}$$

Figure 7: Evaluation rules for monadic reduction. For the sake of readability, type annotations in STM-UNFOLD and STM-PAUSEBIND are elided.

$$\frac{}{\langle \mathsf{nil}, \mathsf{nil}\rangle\ wc\ \langle \mathsf{nil}, \mathsf{nil}\rangle}\ \text{(WC-ID)}$$

$$\frac{\langle \Sigma_1, \Sigma_2\rangle\ wc\ \langle \Sigma_1', \Sigma_2'\rangle}{\langle s_1 :: \Sigma_1, s_2 :: \Sigma_2\rangle wc\ \langle s_1 :: \Sigma_1', s_2 :: \Sigma_2'\rangle}\ \text{(WC-UNCHANGED)}$$

$$\frac{\langle \Sigma_1, \Sigma_2\rangle\ wc\ \langle \Sigma_1', \Sigma_2'\rangle}{\langle s_1 :: \Sigma_1, s_2 :: \Sigma_2\rangle wc\ \langle s :: \Sigma_1', s :: \Sigma_2'\rangle}\ \text{(WC-CHANGED)}$$

Figure 10: The write consistency relation.

does not alter stores where no writes are allowed (Theorem 6); and moreover, that monadic-reduction does not reveal any changes to stores relative to monads with effect labels where only reads are allowed (Theorem 7). To that end, we make use of three relations: "same where no writes", "same where read", and "write consistency", written $\overset{M(\text{W̶})}{=}$, $\overset{M(\mathsf{R})}{=}$, and *wc* – and defined in Figure 8, Figure 9, and Figure 10 – respectively.

Stores (semantically) correspond to state monad transformers. Given a well-typed configuration, the associated store will contain appropriate elements relative to each layer in the state monad transformer stack. In order to update a store, its state monad must contain a write label.

Theorem 6 (No Forbidden Updates): If $\langle t, \Sigma\rangle \rhd M\ \tau$, then $\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma'\rangle$ implies $\Sigma \overset{M(\text{W̶})}{=} \Sigma'$.

Similarly, reading from a store takes place only relative to state monads that have a read label. This is reflected in the type judgments for put (resp., get) that require a write (resp., read) label in order to be well-typed.

Theorem 7 (No Forbidden Reads): Suppose $\Sigma_1 \overset{M(\mathsf{R})}{=} \Sigma_2$ and that $\langle t, \Sigma_1\rangle \rhd M\ \tau$ and $\langle t, \Sigma_2\rangle \rhd M\ \tau$. Then if $\langle t, \Sigma_1\rangle \rightsquigarrow \langle t_1', \Sigma_1'\rangle$ and $\langle t, \Sigma_2\rangle \rightsquigarrow \langle t_2', \Sigma_2'\rangle$, it follows that $t_1' = t_2'$ and $\langle \Sigma_1, \Sigma_2\rangle$ is write consistent with $\langle \Sigma_1', \Sigma_2'\rangle$.

The intuition underlying write consistency is that when considering a pair of stores Σ_1 and Σ_2, prior to a reduction and a pair of matching stores Σ_1' and Σ_2', after a reduction it is either the case that the pre-reduction stores do not differ from their corresponding post-reduction stores (i.e. because no write takes place) or are equal to each other (i.e., because the same value was written to both Σ_1 and Σ_2).

V. Type-directed Equational Logic for ReWire Calculus

The rules provided in Figure 11 represent the properties of monads present in RWC. Rules (LEFT-UNIT), (RIGHT-UNIT), and (ASSOCIATIVITY) are the well-known "monad laws" and Rules (LIFT-RETURN) and (LIFT-$\star$) are the "lifting laws" of Liang [12]. Rules (PUT-PUT), (PUT-GET), and (GET-GET), specify the interaction of stateful operations and are drawn from previous work [5]. The $\leq$ relation on state monads is defined in Figure 4. We defer discussion of the remaining rules until the next section.

The equational logic of RWC has both atomic noninterference and clobber formalized as consequences of the RWC semantics in Coq; here, we refer to the last three rules of Figure 11. These are particular instances for a two layer state monad of the more general rules found in the Coq script repository. Note that, in its Coq formalization, mask computes the appropriate definition from a monad type term taken as an argument. The exact details of this definition need not concern us here, and the interested reader may consult the repository.

VI. Conclusions and Future Work

The ReWire methodology differs fundamentally from the type-based approach to secure hardware (e.g., that of Caisson [17], Sapper [18], and SecVerilog [19]) in three important respects. Firstly, ReWire is a functional language (a subset of Haskell) and has the benefit, we would argue, of the expressiveness of functional languages. Secondly, ReWire possesses a formal semantics and equational theory mechanized in the Coq theorem proving system, allowing security verification

$$\frac{t = t' : \tau \in \Gamma}{\Gamma \vdash t = t' : \tau}\ (\textsc{Axiom}) \quad \frac{\Gamma \vdash t = t' : \tau}{\gamma, \Gamma \vdash t = t' : \tau}\ (\textsc{Weakening}) \quad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash t = t : \tau}\ (\textsc{Refl}) \quad \frac{\Gamma \vdash t' = t : \tau}{\Gamma \vdash t = t' : \tau}\ (\textsc{Sym}) \quad \frac{\Gamma \vdash t = t' : \tau \quad \Gamma \vdash t' = t'' : \tau}{\Gamma \vdash t = t'' : \tau}\ (\textsc{Trans})$$

$$\frac{y \notin \textsc{fv}(t)}{\Gamma \vdash \lambda x{:}\tau.t = \lambda y{:}\tau.t[x \coloneqq y] : \tau \to \tau'}\ (\alpha) \quad \frac{\Gamma \vdash \lambda x{:}\tau.t : \tau \to \tau' \quad \Gamma \vdash t' : \tau}{\Gamma \vdash (\lambda x{:}\tau.t)t' = t[t' \coloneqq x] : \tau'}\ (\beta)$$

$$\frac{\Gamma \vdash t' : \tau \quad \Gamma \vdash t : \tau \to \mathsf{M}\,\tau'}{\Gamma \vdash (\mathsf{return}_{\mathsf{M}}\, t')\ \texttt{>>=}\ t = t\ t' : \mathsf{M}\,\tau'}\ (\textsc{Left-Unit}) \quad \frac{\Gamma \vdash t : \mathsf{M}\,\tau}{\Gamma \vdash t\ \texttt{>>=}\ \lambda x{:}\tau.(\mathsf{return}_{\mathsf{M}}\, x) = t : \mathsf{M}\,\tau}\ (\textsc{Right-Unit})$$

$$\frac{\Gamma \vdash t : \mathsf{M}\,\tau \quad \Gamma \vdash t' : \tau \to \mathsf{M}\,\tau' \quad \Gamma \vdash t'' : \tau' \to \mathsf{M}\,\tau'' \quad x \notin FV(t')}{\Gamma \vdash (t\ \texttt{>>=}\ t')\ \texttt{>>=}\ t'' = t\ \texttt{>>=}\ (\lambda x{:}\tau.t'x\ \texttt{>>=}\ t'') : \mathsf{M}\,\tau''}\ (\textsc{associativity-}\star)$$

$$\frac{\Gamma \vdash \mathsf{return}_{\mathsf{M}}\, t : \mathsf{M}\,\tau}{\Gamma \vdash \mathsf{lift}_{\mathsf{M}'}(\mathsf{return}_{\mathsf{M}}\ t) = \mathsf{return}_{\mathsf{M}'}\ t : \mathsf{M}'\,\tau}\ (\textsc{Lift-Return}) \quad \frac{\Gamma \vdash t : \mathsf{M}\,\tau \quad \Gamma \vdash t' : \tau \to \mathsf{M}\,\tau' \quad x \notin FV(t')}{\Gamma \vdash \mathsf{lift}_{M}\,(t\ \texttt{>>=}\ t') = (\mathsf{lift}_{M}\ t)\ \texttt{>>=}\ (\lambda(x:\tau).\,\mathsf{lift}_{M}\ (t'\,x))}\ (\textsc{Lift-}\star)$$

$$\frac{\Gamma \vdash \mathsf{put}\ t : \mathsf{StT}\,\ell\,\tau\,\mathsf{S}\,() \quad \Gamma \vdash \mathsf{put}\ t' : \mathsf{StT}\,\ell\,\tau\,\mathsf{S}\,()}{\Gamma \vdash (\mathsf{put}\ t\ \texttt{>>}\ \mathsf{put}\ t') = \mathsf{put}\ t' : \mathsf{StT}\,\ell\,\tau\,\mathsf{S}\,()}\ (\textsc{Put-Put})$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash (\mathsf{put}\ t\ \texttt{>>}\ \mathsf{get}_{(\mathsf{StT\,RW}\,\tau\,\mathsf{S})}) = \mathsf{put}\ t\ \texttt{>>}\ \mathsf{return}_{(\mathsf{StT\,RW}\,\tau\,\mathsf{S})}\ t : \mathsf{StT\,RW}\,\tau\,\mathsf{S}\,\tau}\ (\textsc{Put-Get})$$

$$\frac{\langle \mathsf{R} \rangle \leq \ell,\ \text{such that } \mathsf{M} = \mathsf{StT}\,\ell\,\tau\,\mathsf{S}}{\Gamma \vdash \mathsf{get}_{\mathsf{M}}\ \texttt{>>=} \lambda x{:}\tau.\,\mathsf{get}_{\mathsf{M}}\ \texttt{>>=} \lambda y{:}\tau.\,\mathsf{return}_{\mathsf{M}}\langle x, y\rangle = \mathsf{get}_{\mathsf{M}}\ \texttt{>>=} \lambda z{:}\tau.\,\mathsf{return}_{\mathsf{M}}\langle z, z\rangle : \mathsf{M}\ (\tau \times \tau)}\ (\textsc{Get-Get})$$

$$\frac{\Gamma \vdash t\colon \mathsf{S}\,\tau \quad \mathsf{S} \text{ is } \mathsf{StT\,RW}\,\tau\,(\mathsf{StT}\langle\rangle\,\tau'\,\mathsf{Id})}{\Gamma \vdash t\ \texttt{>>}\ (\mathsf{mask\,S}) = \mathsf{mask\,S} : \mathsf{S}\,()}\ (\textsc{Clobber-Lo}) \quad \frac{\Gamma \vdash \mathsf{lift}_S\ t\colon \mathsf{S}\,() \quad \mathsf{S} \text{ is } \mathsf{StT}\,\langle\rangle\,\tau\,(\mathsf{StT\,RW}\,\tau'\,\mathsf{Id})}{\Gamma \vdash \mathsf{lift}_S\ t\ \texttt{>>}\ (\mathsf{mask\,S}) = (\mathsf{mask\,S})\ \texttt{>>}\ \mathsf{lift}_S\ t : \mathsf{S}\,()}\ (\textsc{Atomic Noninterference})$$

Figure 11: Type-directed Equational Logic for RWC

to be automatically checked with the attendant increased assurance. Thirdly, and most importantly, ReWire's type system is not a security type system in the usual sense [34]. Security verification in ReWire is not fully automatic via a security type system, but, rather, the equational style of security verification of our previous work [1], [5] is supported by an effects type system based on the marriage of effects and monads [6]. However, we believe that ReWire's being a pure functional language will support the adaptation of ideas from language-based security to the construction of high assurance, secure hardware via extensions to the ReWire type system.

The ReWire methodology, therefore, occupies a middle ground between the security-via-typechecking approach of Caisson and SecVerilog and traditional hardware verification with theorem provers [20]. It combines the advantages of both—static checking on the one hand and deductive reasoning on the other—with the expressive power of functional languages. Our previous work [1]–[3] demonstrates that ReWire possesses what the creators of the Delite framework refer to as "the three P's" [35]: productivity, performance and portability. The current work shows ReWire also possesses a fourth "P": provability. Follow-on articles will present the formalizations of previously published verifications of ReWire devices [1], [2].

The CompCert [36] project mechanizes both a source language's semantics and compiler in Coq, thereby providing the foundation for (1) verifying properties of C source programs and (2) compiling those programs to efficient implementations in a verifiably property-preserving manner. One particular strength of the CompCert approach is that other tools may be mechanized in Coq as well (e.g., static analysis tools, etc., from the Verified Software Toolchain [37]) to provide increased automation and trust to the whole workflow. The current work is motivated by the goal of producing trusted hardware in the same manner as CompCert supports trusted C implementations. This is, admittedly, a very ambitious goal, but the current work is an early, yet important, step in this program. The current work also provides an important first step towards the formal verification of the ReWire compiler.

References

[1] A. Procter, W. Harrison, I. Graves, M. Becchi, and G. Allwein, "A principled approach to secure multi-core processor design with ReWire," *ACM TECS*, vol. 16, no. 2, pp. 33:1–33:25, Feb. 2017.

[2] I. Graves, W. Harrison, A. Procter, and G. Allwein, "Provably correct development of reconfigurable hardware designs via equational reasoning," in *IEEE Inter. Conf. on Field-Programmable Technology (ICFPT)*, 2015, pp. 160–171.

[3] I. Graves, A. Procter, W. Harrison, M. Becchi, and G. Allwein, "Hardware synthesis from functional embedded domain-specific languages: A case study in regular expression compilation," in *Applied Reconfigurable Computing*, ser. LNCS, vol. 9040, 2015, pp. 41–52.

[4] W. Harrison, A. Procter, I. Graves, M. Becchi, and G. Allwein, "A programming model for reconfigurable computing based in functional concurrency," in *11th Inter. Symp. on Reconfigurable Communication-centric Systems-on-Chip*, 2016.

[5] W. Harrison and J. Hook, "Achieving information flow security through monadic control of effects," *JCS*, vol. 17, pp. 599–653, Oct 2009.

[6] P. Wadler, "The marriage of effects and monads," in *ICFP*, 1998, pp. 63–74.

[7] A. Procter, "Semantics-driven design and implementation of high-assurance hardware," Ph.D. dissertation, University of Missouri, 2014.

[8] B. Huffman, "HOLCF '11: A definitional domain theory for verifying functional programs," Ph.D. dissertation, Portland State University, 2012.

[9] L. Schröder and T. Mossakowski, "Hascasl: Integrated higher-order specification and program development," *Theoretical Computer Science*, vol. 410, no. 12, pp. 1217 – 1260, 2009.

[10] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, July 1991.

[11] F. Nielson, H. Nielson, and C. Hankin, *Principles of Program Analysis*, 1999.

[12] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *POPL*, 1995, pp. 333–343.

[13] A. P. W. Harrison and G. Allwein, "The confinement problem in the presence of faults," in *ICFEM*, 2012, pp. 182–197.

[14] D. Cock, G. Klein, and T. Sewell, "Secure microkernels, state monads and scalable refinement," in *TPHOLs*, 2008, pp. 167–182.

[15] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal, "Ynot: Dependent types for imperative programs," in *ICFP*, 2008, pp. 229–240.
[16] W. Swierstra, "A hoare logic for the state monad," in *TPHOLs*, 2009, pp. 440–451.
[17] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *PLDI*, 2011, pp. 109–120.
[18] X. Li, V. Kashyap, J. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. Chong, "Sapper: A language for hardware-level security policy enforcement," in *ASPLOS*, 2014.
[19] D. Zhang, Y. Wang, G. E. Suh, and A. Myers, "A hardware design language for efficient control of timing channels," Dept. of Computer Science, Cornell University, Tech. Rep. 2014-04-10, 2014, extended version of the authors' ASPLOS15 paper.
[20] T. Melham, *Higher Order Logic and Hardware Verification*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993, vol. 31.
[21] G. Cabodi and M. Murciano, "BDD-based hardware verification," in *6th Inter. Conf. on Formal Methods for the Design of Computer, Communication, and Software Systems*, ser. SFM'06, 2006, pp. 78–107.
[22] A. Myers, personal communication, Mar. 2017.
[23] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *POPL*, 2014, pp. 165–178.
[24] M. Gordon, "The semantic challenge of Verilog HDL," in *Logic in Computer Science, 1995. LICS '95. Proceedings., Tenth Annual IEEE Symposium on*, Jun 1995, pp. 136–145.
[25] C. Kloos and P. Breuer, Eds., *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.
[26] S. Goncharov and L. Schröder, "A coinductive calculus for asynchronous side-effecting processes," in *Proc. of the 18th International Conf. on Fundamentals of Computation Theory*, 2011, pp. 276–287.
[27] K. Crary, A. Kliger, and F. Pfenning, "A monadic analysis of information flow security with mutable state," *JFP*, vol. 15, no. 2, pp. 249–291, Mar. 2005.
[28] D. Ghica and A. Jung, "Categorical semantics of digital circuits," in *FMCAD*, 2016.
[29] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.
[30] W. Harrison, "The essence of multitasking," in *Algebraic Methodology and Software Technology*, 2006, pp. 158–172.
[31] J. Goguen and J. Meseguer, "Unwinding and inference control," in *IEEE Symp. on Security and Privacy*, 1984, pp. 75–86.
[32] J. Mitchell, *Foundations for Programming Languages*. MIT Press Cambridge, 1996.
[33] D. Sangiorgi, "On the origins of bisimulation and coinduction," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 4, pp. 15:1–15:41, May 2009.
[34] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journ. on Sel. Areas in Commun.*, vol. 21, no. 1, Jan. 2003.
[35] 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*, vol. 31, no. 5, pp. 42–53, Sep. 2011.
[36] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009.
[37] "Verified Software Toolchain," http://vst.cs.princeton.edu.