The Mechanized Marriage of Effects and Monads with Applications to High Assurance Hardware

THOMAS N. REYNOLDS, University of Missouri
ADAM PROCTER*, University of Missouri
WILLIAM L. HARRISON, University of Missouri
GERARD ALLWEIN, US Naval Research Laboratory, USA

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.

CCS Concepts: • **Security and privacy** → *Logic and verification*; • **Computer systems organization** → *Embedded hardware*;

Additional Key Words and Phrases: High-level Synthesis; Hardware Verification; Security

ACM Reference Format:
Thomas N. Reynolds, Adam Procter, William L. Harrison, and Gerard Allwein. 2018. The Mechanized Marriage of Effects and Monads with Applications to High Assurance Hardware. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 0 (March 2018), 25 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

It is generally recognized that reconfigurable technology has a "programmability" problem [1, 5] and high-level synthesis (HLS) from functional languages is a commonly proposed remedy for this problem [3, 4, 8, 19, 20, 33, 66, 82]. Pure functional languages—i.e., those without side effects—support equational reasoning as a basis for program verification. Combining the two—i.e., HLS from a pure functional language—provides a methodology for high assurance hardware as demonstrated in previous work by the authors [27, 28, 32, 58]. The current article addresses the formalization of this methodology by mechanizing the semantics for a pure HLS language—namely, ReWire—in the Coq theorem proving system [14], with the goal of combining the programmability advantages of

*Current affiliation: Intel Nervana, San Diego California, USA.

Authors' addresses: Thomas N. Reynolds, University of Missouri, Department of Computer Science, thomas.reynolds@mail.missouri.edu; Adam Procter, University of Missouri, Department of Computer Science, adamprocter@mail.missouri.edu; William L. Harrison, University of Missouri, Department of Computer Science, harrisonwl@missouri.edu; Gerard Allwein, US Naval Research Laboratory, Washington DC, USA.

1539-9087/2018/3-ART0 $15.00
https://doi.org/0000001.0000001

functional hardware description with formalized reasoning. All of the definitions and theorems in this paper have been checked with the Coq proof checker; the Coq v8.5 scripts are downloadable [59].

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 [27, 28, 32, 58]. This article describes the Coq formalization of ReWire intended to support the verification of hardware designs and, in particular, the information flow properties described in our previous work [27, 30, 58].

The ReWire development flow is intended to approach that of functional programming to the greatest extent possible. First, device specifications are "roughed out" in Haskell, allowing testing and debugging in a familiar mode (e.g., using QuickCheck [12] as we did in Graves et al. [27]). Formal specification and verification typically starts at this point in the process. Refactoring into ReWire generally involves choosing base types (i.e., replacing Haskell's `Data.Word` with ReWire's built-in types). The ReWire compiler produces VHDL and vendor tools are used to synthesize, etc., to an FPGA. Making a formal methodology out of this requires a mechanized semantics for ReWire as a foundation for verification of designs and of the ReWire compiler. This article provides that foundation.

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 2 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 and mechanizing Wadler's "marriage" of effects and monads [80]) 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 ⟨⟩ Lo Id)` (resp., `StT ⟨⟩ Hi (StT RW Lo Id)`). The effect label "`RW`" means *h* can both read and write on the `Hi` layer and while "`⟨⟩`" means it can do neither on the `Lo` layer (and, *vice versa*, for *l*). The soundness of our type system (Theorems 4.13 and 4.15) guarantees freedom from covert storage channels. **(2)** A small-step semantics for ReWire formalized in Coq that justifies **(3)** a typed equational logic (Figure 14) 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.

The direct approach to formalizing ReWire in Coq would 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 2) for 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 [56], building on existing work by Huffman [34] or Schröder and Mossakowski [65] 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 2 presents an overview of ReWire to motivate the formal calculus, RWC. Section 3 defines the syntax and small-step operational semantics of RWC. Section 4 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 5. Section 6 discusses conclusions and future work.

Related Work

Andrews [1] argues that a paradigm shift for reconfigurable computing is a necessary precondition for wider adoption of reconfigurable technology. Rather than focusing exclusively on performance metrics, the new paradigm must focus as well on what, for lack of a better term, might be called software engineering virtues—abstraction, modularity, program comprehensibility, productivity, rapid modifiability, reuse, and scalability, etc. What is required are programming models/languages for reconfigurable computing that embrace the software engineering virtues.

One proposed remedy to the programmability issue is high-level synthesis from functional languages [19], because, as originally observed by Sheeran [66], combinational logic has a functional flavor. More to the point, functional languages support the software engineering virtues through higher-order abstractions and type systems. ReWire provides the usual functional programming model of combinational logic—i.e., pure functions—but it also provides a formal model of synchronous logic in the form of the reactive resumption monad discussed in Section 2.

There are a number of efforts to apply ideas and techniques from functional programming to hardware design and synthesis. Chisel [4] is a Scala-embedded domain-specific language developed as an implementation language for the RISC-V open source instruction set architecture[1]. Within the Haskell community, perhaps the most well known system for hardware synthesis is Lava [8]. Lava is a domain-specific language for hardware specification embedded in Haskell. Primitives in Lava are essentially structural and specify circuits at the level of signals. ReWire, by contrast, compiles a subset of Haskell itself to hardware circuits, and relies on an abstract set of behavioral primitives. The primary motivation for developing ReWire is as a vehicle for the design, implementation, and formal verification of high assurance hardware. There are some constructs of VHDL that have not been implemented in ReWire (e.g., tri-state buffers, multiple clock domains, etc.). We believe such constructs can be readily modeled in ReWire, but they have not been necessary for previous case studies [27, 28, 31, 32, 57, 58].

ForSyDe (Formal System Design)[2] is a formal design methodology that targets heterogeneous embedded systems [62, 63]. The ForSyDe toolset includes a system modeling language implemented as an embedded domain-specific language in Haskell that contains elements similar to those in ReWire, albeit not in resumption-monadic form. The ForSyDe methodology is based on refinement: high-level models are transformed semi-automatically into heterogeneous (i.e., mixed hardware and software) embedded systems. The ReWire methodology differs from that of ForSyDe in a number of respects. The ReWire language has type constructors for devices (described below in Section 2) that are compiled automatically into VHDL by the ReWire compiler, so hardware is generated directly from ReWire source code rather than produced by semi-automatic refinement. ForSyde targets heterogeneous hardware and software systems whereas ReWire focuses on hardware exclusively. Finally, the formal methodology supported by ReWire, illustrated in previous publications [27, 31, 57, 58], is precisely that of pure functional languages; this is sometimes referred to as "Bird-Wadler"

[1]https://riscv.org.
[2]https://forsyde.ict.kth.se/trac.

style program derivation (so-named after an influential textbook [7]). A Bird-Wadler derivation starts from a reference specification for an algorithm in a functional language and, through a series of semantics-preserving program transformations, produces a more efficient implementation. Desired properties of the implementation (e.g., correctness, security, etc.) are specified equationally and verified in terms of the reference semantics. The current work represents the formalization of the ReWire methodology in Coq.

Zhai et al. [82] consider high-level synthesis from recursive functional languages. Similarly, Cλash [3], is a compiler for a subset of Haskell to VHDL. Like ReWire, Cλash uses Haskell itself as a source language. Cλash requires some limits be placed on the kinds of algebraic data types used as well as the basic operating types. Both differ fundamentally from ReWire in that they require that a stack be constructed in hardware as part of the circuits they produce. It was an early design decision in the ReWire project to limit recursive functions to co-recursion (tail recursion) so as to obviate the need for a run-time stack or other unbounded data structures. Given hardware's fixed memory footprint, it seemed more natural to us to not require support for potentially unbounded data. Great care was taken in the design of ReWire so that it possesses a rigorous denotational semantics to support formal verification while maintaining synthesizability for all of its programs [56].

The Delite DSL compiler framework [39] seeks to address the "three P's" with respect to implementing software on parallel, heterogeneous systems. Delite addresses portability (i.e., retargetability of DSL compilers to a broad range of parallel hardware) through *language virtualization.* ReWire is also a virtualized DSL in that it has a separate compiler backend for producing FPGA-based implementations while reusing large parts of its host language's infrastructure—including Haskell's type system, front end, etc. In George, et al., [20], the Delite framework is adapted to the generation of hardware from DSLs, specifically the hardware acceleration of kernels in a heterogeneous setting.

There is a vast literature on hardware security from an architectural or physical perspective, considering issues ranging from side channel attacks, hardware trojan detection, and the like. For an overview of this literature, please consult the references [35, 71, 76]. The architectural perspective of hardware security considers hardware structures or designs supporting security policies. To take one example from among many, GLIFT [74] is a gate-level information flow tracking method that inserts special "shadow circuits" to dynamically monitor all information flows within a circuit. The references include other examples of this architectural perspective [6, 36, 37, 67, 72, 73, 75, 81]. There is an orthogonal line of research in hardware security that considers the design, implementation and formal verification of hardware from a languages-based approach which we overview below; ReWire fits squarely within this research thrust.

Formal methods for secure hardware are generally spread across two categories: (1) type-based approaches [41, 42, 83]; and (2) logic-based approaches (including theorem-proving [45], and BDDs and model-checking [10]), 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. With any security type system, the question of its expressiveness arises—i.e., does it reject secure designs? The types-based approach offers no recourse to the rejection of a secure design—you simply cannot argue with a type checker. A logic-based approach avoids this pitfall, but comes with overhead—e.g., your own theory of security—and neither is it connected directly to any implementation path.

Bluespec [33, 52] refers to a language and associated tools for hardware system design, specification, synthesis, modeling, and verification. There have been a number of incarnations of the Bluespec language since its inception in 2000, the first of which was as a Haskell subset extended with domain-specific operations for hardware design. The Bluespec language seems to have evolved into its current form which is BSV (Bluespec SystemVerilog), which is no longer a functional language. There have been some formal methods tools developed for BSV [53, 60].

Braibant and Chlipala [9] apply ideas from CompCert [40] to hardware synthesis and is the most closely related to our own. Their work presents a certified compiler translating a monadic-functional HDL (called "Fe-Si") into RTL. Fe-Si is a small, idealized core of the BSV hardware description language [33]. Fe-Si's syntax is based on state monads, albeit not structured with monad transformers like ReWire's. Timing in Fe-Si is explicit, rather in the manner of VHDL, using an explicit clock tick parameter, whereas ReWire makes use of reactive resumptions as a basis for timing (see Section 2.2.2 below). One of the primary motivations behind the current work is to build a foundation for a verified compilation process for ReWire. Choi et al. [11] follow Braibant and Chlipala's work, starting from an idealized, BlueSpec-like language.

One language-based approach to hardware security is to extend an existing HDL with security types. Caisson [42], Sapper [41], and SecVerilog [83] 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 [49]. 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 6.

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 [2], 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 [45]. 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 [26, 38]. 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 [27, 58], we presented several case studies in hardware verification based in ReWire, but there the verifications were not machine-checked.

Goncharov and Schröder [25] 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. [16] 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 [30] is more semantic and model-theoretic than Crary's logical and type-theoretic approach, relying on structural properties of monads and monad transformers to construct secure systems. Security verifications of ReWire designs [58] are based on Harrison and Hook's approach, and the present work formally supports that approach in Coq.

Ghica and Jung [21] 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 [27, 58]. By contrast with Ghica and Jung's work, ReWire specifications are, more or less, ordinary functional programs that are compiled into circuits. Another categorical presentation of digital circuits is found in Megacz [44], who uses generalized arrows as a basis for hardware description.

Effect systems are a static semantics of effects while monads [48] are a dynamic semantics of effects. Effect systems [51] were initially 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 [43]—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 [80] "married" effect types to monads, and previous work by the authors [79] 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 14).

As a concept for formal (i.e., machine-checked) verification, monads are less common, although not unheard of [13, 50, 65, 68] and the use of both effect types and layered monads distinguishes the current work from these. Furthermore, ReWire's monad language includes the reactive resumption monad transformer, which does not appear to have been formalized previously.

2 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.

ReWire is a subset of the Haskell functional programming language [54]—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 a stack or heap for compilation and such unbounded 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 Fig. 1). 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 [32] introduced operators for constructing devices and

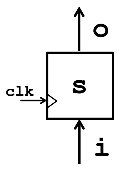

Fig. 1. Device d

composing them into larger, interconnected devices; Section 2.3 presents a simple device specification template in ReWire.

2.1 Background: Monads

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

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

These operations must obey the well-known *monad laws* [43, 48] (these are (Left-Unit), (Right-Unit), and (Associativity) in Figure 14). 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, $\mathtt{>>} : \mathtt{M\,a} \rightarrow \mathtt{M\,b} \rightarrow \mathtt{M\,b}$, is defined as: $\mathtt{x} \mathbin{>>} \mathtt{k} = \mathtt{x} \mathbin{>>=} \lambda\_.\mathtt{k}$. The binding (i.e., “$\lambda\_$”) acts as a dummy variable, ignoring the value produced by x.

2.2 Background: Monad Transformers

The organizing principle underlying ReWire are *reactive resumption monads with state* [29] (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 [32]. 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 [29, 58].

2.2.1 *State Monad Transformer.*

The state monad transformer is a well-documented structure in functional programming and semantics [43]. 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 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.

2.2.2 *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 [58]. 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. 3) as a means of representing `signal`.

2.2.3 By-construction Properties of Layered Monads. Layered state monads have multiple `StT` applications—e.g., $\mathtt{M} = \mathtt{StT}\ s_1\ (\mathtt{StT}\ s_2\ \mathtt{Id})$ is a two-layer state monad. They have a number of useful properties by construction [30], including:

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

The first rule is an intra-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 $\mathtt{mask} = \mathtt{put}\ s_0$; 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 5 gives generalizations of both properties.

2.3 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 functions, $\mathtt{internal} :: \mathtt{i} \rightarrow \mathtt{StT\ s\ Id\ v}$ and $\mathtt{external} :: \mathtt{i} \rightarrow \mathtt{v} \rightarrow \mathtt{o}$, are defined which specify the internal and external behaviors of device d. 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 i_0, $\mathtt{d} = \mathtt{dev}\ i_0$ where corecursive function dev is defined as:

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

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

Device definitions are expressed with an explicit corecursion operator, `unfold`; for example, the device dev above would be written:

```
unfold i_0 (\ i -> internal i >>= \ v -> return (Right (external i v, id)))
```

For this reason, Figure 3 includes syntax for an `unfold` primitive and its semantics are defined in subsequent sections.

2.4 Background: Goguen-Meseguer Non-interference

The essence of the Goguen-Meseguer noninterference information flow model [24] 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, $\mathsf{s} = (\mathtt{l}_1 \,;\, \mathsf{h}_1 \,;\, \ldots \,;\, \mathtt{l}_n \,;\, \mathsf{h}_n)$, 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.

2.5 Marrying Effects & Layered State Monads

"By construction" properties of layered state monads [30] 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" [30]. 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 [30, 58].

The Goguen-Meseguer model was recast in monadic terms previously [30], 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:

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

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

3 RWC: THE REWIRE CORE CALCULUS

This section introduces the syntax (Section 3.1), type system (Section 3.2) and operational semantics (Section 3.3) 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* [48], 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 [80].

3.1 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 [30, 58]; we defer presenting the general formalization of these by-construction properties until Section 5. Section 2 provides the reader with some background on monad transformers, although readers requiring more should consult the references.

3.1.1 Types. Figure 2 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*

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

Fig. 2. Syntax of RWC types

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

Fig. 3. Syntax of terms, stores, and configurations

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 [29] 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 [56] for further details.

3.1.2 Terms. Figure 3 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 >>= 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 5). 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 runIdallow 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 runIdas moving from the effect-free Id monad into the universe of non-monadic terms.

3.1.3 Stores and Configurations. Figure 3 (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.

3.2 Type System

Typing rules for terms are given in Figure 4. 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 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 5 at left. For T-Elevate, we require that the target monad S' has (non-strictly) more effect labels than the source monad S; its precise meaning is expressed in Figure 5 at right. The elevate operation 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 6. 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 $\langle t, \Sigma\rangle \triangleright M\ \tau$. A simple, yet useful property of our type system is that every term (resp. configuration) has a unique type, as stated in Theorem 3.1.

Theorem 3.1 (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 $\tau = \tau'$.*

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ (\textsc{Var}) \quad \frac{\Gamma, x:\tau \vdash t:\tau'}{\Gamma \vdash \lambda x:\tau.t:\tau \rightarrow \tau'}\ (\textsc{Abs}) \quad \frac{\Gamma \vdash t':\tau \rightarrow \tau' \quad \Gamma \vdash t:\tau}{\Gamma \vdash t'\ t:\tau'}\ (\textsc{App})$$

$$\frac{\Gamma \vdash t:\tau}{\Gamma \vdash \mathsf{inl}_{\tau'}\ t:\tau+\tau'}\ (\textsc{Inl}) \quad \frac{\Gamma \vdash t:\tau}{\Gamma \vdash \mathsf{inr}_{\tau'}\ t:\tau'+\tau}\ (\textsc{Inr}) \quad \frac{\Gamma \vdash t:\tau \quad \Gamma \vdash t':\tau'}{\Gamma \vdash \langle t, t'\rangle:\tau\times\tau'}\ (\textsc{Pair})$$

$$\frac{}{\Gamma \vdash ():()}\ (\textsc{Unit}) \quad \frac{\Gamma \vdash t:\tau\times\tau' \quad \Gamma \vdash t':\tau \rightarrow \tau' \rightarrow \tau''}{\Gamma \vdash \mathsf{proj}\ t\ t':\tau''}\ (\textsc{Proj})$$

$$\frac{\Gamma \vdash t:\tau'+\tau'' \quad \Gamma \vdash t':\tau' \rightarrow \tau \quad \Gamma \vdash t'':\tau'' \rightarrow \tau}{\Gamma \vdash \mathsf{case}\ t\ t'\ t'':\tau}\ (\textsc{Case})$$

$$\frac{\Gamma \vdash t:\tau}{\Gamma \vdash \mathsf{return}_{\mathsf{M}}\ t:\mathsf{M}\ \tau}\ (\textsc{Return}) \quad \frac{\Gamma \vdash t:\mathsf{M}\ \tau \quad \Gamma \vdash t':\tau \rightarrow \mathsf{M}\ \tau'}{\Gamma \vdash t\ \mathsf{>>=}\ t':\mathsf{M}\ \tau'}\ (\textsc{Bind})$$

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

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

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

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

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

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

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

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

Fig. 4. Typing judgments for terms.

Furthermore, substitutions preserve typing judgments. To see this, we need to define substitution and collect some facts about free variables, substitutions and types. For any term t, the set of free variables in t, $FV(t)$, is defined as follows:

$$\begin{array}{rcll}
FV(x) & = & \{x\} & \\
FV(t\,u) & = & FV(t) \bigcup FV(u) & \\
FV(\lambda x:\tau.t) & = & FV(t) \setminus \{x\} & \\
FV(c_0) & = & \{\}, & \text{for any nullary term } c_0 \\
FV(c_n\, t_1, \ldots, t_n) & = & \bigcup_{i=1}^{n} FV(t_i), & \text{for an } n\text{-ary term } c_n
\end{array}$$

In the last clause above, the c in '$c\, t_1, \ldots, t_n$' stands for term constructors such as case, return$_M$, etc. If $FV(t) = \emptyset$, then t is said to be *closed.* We now define the substitution of v for free occurrences of x in t, written '$t[x := v]$', thusly:

$$\begin{array}{rcll}
x[x := v] & = & v & \\
y[x := v] & = & y & \text{if } y \neq x \\
(t\,u)[x := v] & = & (t[x := v])(u[x := v]) & \\
(\lambda x:\tau.t)[x := v] & = & (\lambda x:\tau.t) & \\
(\lambda y:\tau.t)[x := v] & = & \lambda y:\tau.(t[x := v]) & \text{if } y \neq x \text{ and } y \notin FV(v) \\
(c\, t_1, \ldots, t_n)[x := v] & = & (c\,(t_1[x := v]), \ldots, (t_n[x := v])) &
\end{array}$$

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

Fig. 5. Ordering on effect labels (given by the diagram) and on state monads.

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

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

Fig. 6. Typing judgments for stores (top) and configurations (bottom).

This definition of substitution preserves typing judgments. This requires that if free variables occur in well-typed terms, then there must be a typing assignment for those variables relative to the context. As stated in Lemma 3.2.

LEMMA 3.2. *If $x \in FV(t)$ and $\Gamma \vdash t : \tau$, then there exists τ' such that $x : \tau' \in \Gamma$.*

From this Corollary 3.3 follows—namely, that a term is closed if it it is well-typed in the empty context.

COROLLARY 3.3. *If $\{\} \vdash t : \tau$, then t is closed.*

Moreover, we have Lemma 3.4 as a consequence—that the context of a typing judgment does not alter typing judgments, so long as all each context maintains assignments of types to any free variables.

LEMMA 3.4. *If $\Gamma \vdash t : \tau$ and, if, for all x such that $x \in \mathbf{FV}(t), \Gamma, x = \Gamma', x$, then $\Gamma' \vdash t : \tau$.*

Finally, we have Theorem 3.5—that is, it follows that substitution preserves typing judgments.

THEOREM 3.5. *If $x : \tau', \Gamma \vdash t : \tau$ and $\{\} \vdash v : \tau'$, then $\Gamma \vdash (t[x := v]) : \tau$.*

3.3 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 7 and 8, 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.As stated in Theorem 3.6, the reduction relation defined by the rules for pure and effectful reduction is deterministic.

$$\frac{}{(\lambda x : \tau . t) v \rightsquigarrow t[x := v]}\ (\text{ST-AppAbs}) \qquad \frac{t \rightsquigarrow t''}{t\ t' \rightsquigarrow t''\ t'}\ (\text{ST-App1}) \qquad \frac{t \rightsquigarrow t'}{v\ t \rightsquigarrow v\ t'}\ (\text{ST-App2}) \qquad \frac{t \rightsquigarrow t''}{\langle t, t' \rangle \rightsquigarrow \langle t'', t' \rangle}\ (\text{ST-Pair1})$$

$$\frac{t \rightsquigarrow t'}{\langle v, t \rangle \rightsquigarrow \langle v, t' \rangle}\ (\text{ST-Pair2}) \qquad \frac{t \rightsquigarrow t'}{\mathsf{proj}\ t\ t'' \rightsquigarrow \mathsf{proj}\ t'\ t''}\ (\text{ST-Proj1}) \qquad \frac{t \rightsquigarrow t'}{\mathsf{proj}\ v\ t \rightsquigarrow \mathsf{proj}\ v\ t'}\ (\text{ST-Proj2})$$

$$\frac{}{\mathsf{proj}\ \langle v, v' \rangle\ v'' \rightsquigarrow (v''\ v)\ v'}\ (\text{ST-Proj}) \qquad \frac{t \rightsquigarrow t'}{\mathsf{inl}_T\ t \rightsquigarrow \mathsf{inl}_T\ t'}\ (\text{ST-Inl}) \qquad \frac{t \rightsquigarrow t'}{\mathsf{inr}_T\ t \rightsquigarrow \mathsf{inr}_T\ t'}\ (\text{ST-Inr})$$

$$\frac{t \rightsquigarrow t'''}{\mathsf{case}\ t\ t'\ t'' \rightsquigarrow \mathsf{case}\ t'''\ t'\ t''}\ (\text{ST-Case1}) \qquad \frac{t \rightsquigarrow t''}{\mathsf{case}\ v\ t\ t' \rightsquigarrow \mathsf{case}\ v\ t''\ t'}\ (\text{ST-Case2}) \qquad \frac{t \rightsquigarrow t'}{\mathsf{case}\ v\ v'\ t \rightsquigarrow \mathsf{case}\ v\ v'\ t'}\ (\text{ST-Case3})$$

$$\frac{}{\mathsf{case}\ (\mathsf{inl}_T\ v)\ v'\ v'' \rightsquigarrow v'\ v}\ (\text{ST-CaseL}) \qquad \frac{}{\mathsf{case}\ (\mathsf{inr}_T\ v)\ v'\ v'' \rightsquigarrow v''\ v}\ (\text{ST-CaseR}) \qquad \frac{\langle v, \mathsf{nil} \rangle \rightsquigarrow \langle t, \mathsf{nil} \rangle}{\mathsf{runId}\ v \rightsquigarrow \mathsf{runId}\ t}\ (\text{ST-RunIdMo})$$

Fig. 7. Reduction Rules for Lambda Calculus Reduction. These rules specify a call-by-value evaluation strategy on RWC.

THEOREM 3.6. *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 $\langle t', \Sigma' \rangle = \langle t'', \Sigma'' \rangle$.*

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 8 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 7 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.

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 >>=, we should "absorb" what comes to the right of the >>= into the pause's continuation, guaranteeing that we make progress towards a "done" configuration.

4 METATHEORY

In this section we discuss the metatheory of RWC, in particular *type safety* (Section 4.1), *strong normalization* (Section 4.3), and *soundness* of effect labels (Section 4.4).

Type systems-based approaches to language-based security (which seem to have originated with Volpano et al. [77]) usually apply type-soundness arguments to demonstrate the correctness of the type system. This soundness argument follows along these lines: well-typed programs (i.e., programs judged secure by the type system) do not misbehave according to a security model (frequently noninterference-based [24]) defined in terms of the language's semantics. The effect system presented in Section 3 can make fine-grained distinctions about memory accesses and, therefore, the soundness of the effect system is highly relevant to multi-level security. For example, "no write down" may be expressed as the type, StT RW H (StT R L Id)(), because any computation with this type may read or write to the high H state, but may only read from the low L state. Similarly,

$$\frac{t \rightsquigarrow t'}{\langle t, \Sigma\rangle \rightsquigarrow \langle t', \Sigma\rangle}\ \text{(STM-ST)} \qquad \frac{\langle v, \Sigma\rangle \rightsquigarrow \langle t, \Sigma'\rangle}{\langle v \mathrel{>\!>\!=} v', \Sigma\rangle \rightsquigarrow \langle t \mathrel{>\!>\!=} v', \Sigma'\rangle}\ \text{(STM-Bind)} \qquad \frac{}{\langle \mathsf{return}_{\mathsf{M}}\ v \mathrel{>\!>\!=} v', \Sigma\rangle \rightsquigarrow \langle v'\ v, \Sigma\rangle}\ \text{(STM-BindRet)}$$

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

$$\frac{}{\langle \mathsf{lift}_{\mathsf{M}}\ (\mathsf{return}_{\mathsf{M}'}\ v), \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{M}}\ v, \Sigma\rangle}\ \text{(STM-LiftRet)}$$

$$\frac{}{\langle \mathsf{get}_{\mathsf{S}}, s :: \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{S}}\ s, s :: \Sigma\rangle}\ \text{(STM-Get)} \qquad \frac{}{\langle \mathsf{put}\ v, s :: \Sigma\rangle \rightsquigarrow \langle \mathsf{return}_{\mathsf{S}}\ (), v :: \Sigma\rangle}\ \text{(STM-Put)}$$

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

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

$$\frac{\langle v, \Sigma\rangle \rightsquigarrow \langle t, \Sigma'\rangle}{\langle \mathsf{runRe}_{\tau}\ v, \Sigma\rangle \rightsquigarrow \langle \mathsf{runRe}_{\tau}\ t, \Sigma'\rangle}\ \text{(STM-RunRe)} \qquad \frac{}{\langle \mathsf{runRe}_{\tau''}\ (\mathsf{pause}_{(\mathsf{ReT}\ \tau\ \tau'\ \mathsf{S})}\ v), \Sigma\rangle \rightsquigarrow \langle v \mathrel{>\!>\!=} \lambda \mathsf{x}.\mathsf{return}\ (\mathsf{inr}_{\tau''}\ \mathsf{x}), \Sigma\rangle}\ \text{(STM-RunRePause)}$$

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

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

$$\frac{}{\langle (\mathsf{pause}\ v) \mathrel{>\!>\!=} v', \Sigma\rangle \rightsquigarrow \langle \mathsf{pause}\ (v \mathrel{>\!>\!=} \lambda w.\ (\mathsf{proj}\ w\ (\lambda x.\lambda y.\ \mathsf{return}\langle x, \lambda z.(y\ z) \mathrel{>\!>\!=} v'\rangle))), \Sigma\rangle}\ \text{(STM-PauseBind)}$$

Fig. 8. Evaluation rules for monadic reduction. For the sake of readability, type annotations in STM-Unfold and STM-PauseBind are elided.

"no read up" is expressed by $\varphi_H : \mathsf{StT\,RW}\,H\,(\mathsf{StT\ R}\,L\,\mathsf{Id})()$. The type soundness demonstrated in Section 4.4 demonstrates the fidelity of RWC types to its operational semantics.

We shall use $\rightsquigarrow^*$ to denote the reflexive, transitive closure of $\rightsquigarrow$. Thus, we have the following properties of $\rightsquigarrow^*$:

Lemma 4.1. *For all terms t, u, v, and stores $\Sigma, \Sigma', \Sigma''$,*

1. *if $u \rightsquigarrow v$, then $u \rightsquigarrow^* v$. Also, if $\langle u, \Sigma\rangle \rightsquigarrow \langle v, \Sigma'\rangle$, then $\langle u, \Sigma\rangle \rightsquigarrow^* \langle v, \Sigma'\rangle$.*
2. *$u \rightsquigarrow^* u$. Also, $\langle u, \Sigma\rangle \rightsquigarrow^* \langle u, \Sigma\rangle$.*
3. *if $t \rightsquigarrow^* u$ and $u \rightsquigarrow^* v$, then $t \rightsquigarrow^* v$. Also, if $\langle t, \Sigma\rangle \rightsquigarrow^* \langle u, \Sigma'\rangle$ and $\langle u, \Sigma'\rangle \rightsquigarrow^* \langle v, \Sigma''\rangle$, then $\langle t, \Sigma\rangle \rightsquigarrow^* \langle v, \Sigma''\rangle$.*

Moreover, for each single-step reduction rule defined in Figures 7 and 8 from Section 3.3, there exists a corresponding version with $\rightsquigarrow^*$ in place of $\rightsquigarrow$.

4.1 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 (Theorem 4.2), and *preservation*, meaning that reduction preserves the types of terms (and configurations) (Theorem 4.3).

Theorem 4.2 (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 4.3 (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 $\langle t', \Sigma'\rangle \triangleright M\ \tau$.*

Together, then, these properties imply that well-typed programs cannot go wrong—i.e., evaluation of well-typed programs never "gets stuck"— as specified in Definition 4.4.

Definition 4.4 (Stuck). A term (resp. configuration) is *stuck* if it is neither a value (resp. done configuration) nor reducible to some other term (resp. configuration).

That is, reduction of well-typed terms (and configurations) will not generate something that is neither a value (resp. done configuration), nor reducible (Corollary 4.5).[3]

Corollary 4.5 (Soundness). *If* $\{\} \vdash t : \tau$ *and* $t \rightsquigarrow^* t'$, *then it is not the case that* t' *is stuck. Also, if* $\langle t, \Sigma \rangle \triangleright M\ \tau$ *and* $\langle t, \Sigma \rangle \rightsquigarrow^* \langle t', \Sigma' \rangle$, *then it is not the case that* $\langle t', \Sigma' \rangle$ *is stuck.*

Perhaps surprisingly, the addition of computational features does not substantially complicate the proof of type safety when compared to similar proofs for pure λ-calculi.

4.2 Canonical Forms

Proofs of metatheoretic theorems about operational semantics (e.g., the proofs of Theorems 4.2, 4.3 and 4.5) are frequently organized in terms of *canonical forms*—that is, closed, well-typed values. The reason for doing so is simply that it drastically reduces the number of cases to be considered in the proof thereby reducing the verification effort. Our canonical forms come in two varieties—the canonical forms of lambda values and canonical forms for monadic expressions stated in Lemmas 4.6 and 4.7, respectively.

Lemma 4.6. *If* $\{\} \vdash v : \tau$ *and* v *is a value, then*

(1) *if* τ *is* $\tau_1 \rightarrow \tau_2$, *there exists* $x\, u$, *such that* $v = \lambda x : \tau_1 . u$,
(2) *if* τ *is* $\tau_1 \times \tau_2$, *there exists* $t_1 t_2, v = \langle t_1, t_2 \rangle$,
(3) *if* τ *is* $\tau_1 + \tau_2$, *there exists* $t', v = \mathsf{inl}_{\tau_2}\ t'$ *or* $v = \mathsf{inr}_{\tau_1}\ t'$,
(4) *if* τ *is* $()$, $v = ()$,

Lemma 4.7. *If* $\langle v, \Sigma \rangle \triangleright \mathsf{M}\ \tau$ *and* $\langle v, \Sigma \rangle$ *is a done configuration, then*

(1) *if* M *is S, there exists* t', $v = \mathsf{return}_{\mathsf{S}}\ t'$,
(2) *if* M *is* $\mathsf{ReT}\ \tau'\ \tau''\ \mathsf{S}$, *there exists* t', $v = \mathsf{return}_{(\mathsf{ReT}\tau'\tau''\mathsf{S}\ \tau)}\ t'$ *or* $v = \mathsf{pause}_{(\mathsf{ReT}\tau'\tau''\mathsf{S}\ \tau)}\ t'$.

The canonical forms for done configurations—and canonical forms in general—greatly reduce the range of potential cases to consider in proofs. In the case of reactive resumptions, the canonical forms for done configurations reflect a fundamental feature of resumptions—namely, that resumptions consume inputs, producing outputs paired with another resumption.

4.3 Strong Normalization

Normalization, generally speaking, is a claim about the set of possible reduction sequences of terms. A reduction relation $\rightsquigarrow$ for a language is *weakly normalizing* if, and only if, for each term t in the language, there is at least one reduction sequence of terms, $t_0 \rightsquigarrow t_1 \rightsquigarrow \cdots \rightsquigarrow t_{n-1} \rightsquigarrow t_n$, such that $t = t_0$ and t_n is irreducible. A reduction relation $\rightsquigarrow$ is *strongly normalizing* if, and only if, every reduction sequence from term t is a prefix of a reduction sequence ending in an irreducible term. Note that strong normalization implies weak normalization, but not vice versa. Note further that these notions of normalization extend in an obvious way from terms to configurations. The Haskell functional language—or, rather, its notion of reduction—is weakly normalizing, but not strongly normalizing, due to Haskell's default lazy evaluation.

Unlike Haskell, RWC enjoys the property of *strong normalization*. 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 [56, 58]. Strong normalization also makes defined equality easier to work with, as it eliminates the need to account for equality of diverging computations.

[3]Soundness follows by induction over the reduction steps taken. Then, apply Theorem 4.3 to show that reduced term is well-typed, followed by an application of Theorem 4.2 to show that the term is either a value or further reducible.

The proof of strong normalization (Theorem 4.11) uses an adaptation of the standard *logical relations* technique [46]. A standard proof using logical relations has two steps. First, define a type-indexed collection of relations over terms. The construction of each relation proceeds inductively by utilizing definitions at "smaller types". The construction of these relations use either one of two approaches—*saturated sets* [69, 70], or *reducibility candidates* [23][4]. Second, establishing that relative to their respective type, every well-typed term respects the relation. In short, given a property $\mathbf{P}$, a *logical relation*, $\mathcal{R}_{\{\mathrm{T}\in\mathcal{T}\}}$ (with respect to $\mathbf{P}$), is a collection of type-indexed relations such that for every $\mathbf{R}_{\mathrm{T}} \in \mathcal{R}_{\{\mathrm{T}\in\mathcal{T}\}}$, every element $t \in \mathbf{R}_{\mathrm{T}}$, either has, or preserves $\mathbf{P}$.

We say that a term t *halts* if and only if there exists a value v (not necessarily distinct from t), 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 $\langle t, \Sigma\rangle \rightsquigarrow^* D$. The interaction between halting and reduction is characterized by the properties collected in Lemma 4.8, while Lemma 4.9 summarizes properties pertaining to reducibility candidates that were used in the course of proving Theorem 4.11. In the case of strong normalization, halting is the property of interest.

Lemma 4.8. *For all terms u, v and stores Σ, Σ',*

(1) *If $u \rightsquigarrow v$, then u halts if and only if v halts.*
(2) *If $u \rightsquigarrow^* v$, then u halts if and only if v halts.*
(3) *If $\langle u, \Sigma\rangle \rightsquigarrow \langle v, \Sigma'\rangle$, then $\langle u, \Sigma\rangle$ halts if and only if $\langle v, \Sigma'\rangle$ halts.*
(4) *If $\langle u, \Sigma\rangle \rightsquigarrow^* \langle v, \Sigma'\rangle$, then $\langle u, \Sigma\rangle$ halts if and only if $\langle v, \Sigma'\rangle$ halts.*

Lemma 4.9. *For all terms u, v and types τ,*

(1) *If $u \rightsquigarrow v$ and $\mathbf{R}_\tau(u)$, then $\mathbf{R}_\tau(v)$*
(2) *If $u \rightsquigarrow^* v$ and $\mathbf{R}_\tau(u)$, then $\mathbf{R}_\tau(v)$*
(3) *If $\{\} \vdash u : \tau$, $u \rightsquigarrow v$ and $\mathbf{R}_\tau(v)$, then $\mathbf{R}_\tau(u)$*
(4) *If $\{\} \vdash u : \tau$, $u \rightsquigarrow^* v$ and $\mathbf{R}_\tau(v)$, then $\mathbf{R}_\tau(u)$,*
(5) *If $\mathbf{R}_\tau(u)$, then u halts.*

We discuss the details of defining reducibility candidates below.

Lemma 4.10. *Let $v_1, \ldots, v_n$ be values of type $\tau_1, \ldots, \tau_n$, such that $\mathbf{R}_{\tau_i}(v_i)$ for each $i = \{1, \ldots, n\}$. Then, if $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$, then $\mathbf{R}_\tau(t[x_1 := v_1] \ldots [x_n := v_n])$.*

Theorem 4.11 follows from Lemma 4.10 using the empty context—keeping in mind that $\mathbf{R}_\tau(t)$, implies that t halts:

Theorem 4.11 (Strong Normalization). *If $\{\} \vdash t : \tau$, then t halts. Also, if $\langle t, \Sigma\rangle \triangleright M\ \tau$, then $\langle t, \Sigma\rangle$ halts.*

4.3.1 Mechanization. Because resumptions are coinductive, and as such, proving that strong normalization holds for configurations requires the use of coinductive proof principles. This is captured by the definition in Figure 10. This allows the $\mathbf{R}$ property to be appropriately applied over reactive resumption computations in a manner that ensures productivity. The use of coinduction and coinductive proof principles has been attributed to David Park [64]. Coquand [15] provided a formalization of coinductive types in type theory using a syntactic guardedness condition. This was implemented in Coq by Giménez [22]. To our knowledge, no other mechanized proofs of strong normalization for computational λ-calculi exist in Coq. There is a proof of strong normalization for Moggi's computational metalanguage in Isabelle/HOL using the nominal package [17].

We formalize $\mathbf{R}$ in Coq using a `Fixpoint` definition in Figure 9. The straightforward Inductive definition violates Coq's strict positivity requirement—that Inductive definitions cannot have

[4]Though similar, saturated sets and reducibility candidates are not the same. See [18] for a detailed comparison.

```
Fixpoint R (τ:Ty) (t:tm) {struct τ} : Prop := ∅ ⊢ t ∈ τ ∧ halts t ∧
   match τ with
   | τ1 → τ2         => ∀t', R τ1 t' → R τ2 (t,t')
   | τ1 × τ2         => ∃ t1 t2, t ⤳* ⟨t1,t2⟩ ∧ value t1 ∧ value t2 ∧ R τ1 t1 ∧ R τ2 t2
   | τ1 + τ2         => ∃ t', value t' ∧ ((t ⤳* inl_τ2 t' ∧ R τ1 t') ∨ (t ⤳* inr_τ1 t' ∧ R τ2 t'))
   | ()              => True
   | SM τ'           => ∀ Σ, Rsto SM Σ → ∃ t' Σ', (t,Σ) ⤳* (return_SM t',Σ') ∧ value t' ∧ R τ' t' ∧ Rsto SM Σ'
   | ReT τi τo SM τ' => ∀ Σ, Rsto SM Σ → ∃ t' Σ', (t,Σ) ⤳* (t',Σ') ∧ value t' ∧ Rsto SM Σ' ∧
                                                along_react (Rsto SM) (R τi) (R τo) (R τ') (t',Σ')
   end
with Rsto SM (Σ:store) {struct SM} : Prop := store_all_values Σ ∧ store_matches_mo Σ SM ∧
                                              match SM with
                                              | Id          => True
                                              | StT ℓ τ SM' => ∃ t Σ', R τ t ∧ Rsto SM' Σ' ∧ Σ = t::Σ'
                                              end.

Inductive store_all_values : store → Prop :=
  | sav_empty : store_all_values ()
  | sav_cons  : ∀ s Σ, value s → store_all_values Σ → store_all_values (s::Σ).

Inductive store_matches_mo : store → Mo → Prop :=
  | matches_mo_id  : store_matches_mo () Id
  | matches_mo_stt : ∀ SM E τ t Σ, ∅ ⊢ t ∈ τ → store_matches_mo Σ SM → store_matches_mo (t::Σ) (StT E τ SM)
  | matches_mo_ret : ∀ SM τi τo Σ, store_matches_mo Σ SM → store_matches_mo Σ (ReT τi τo SM).
```

Fig. 9. The fixpoint definition of logical relation **R**. The store_matches_mo formalizes the *matches* relation from Fig. 6 in Coq.

constructors occurring to the left of an arrow [55]. The core of the definition for ordinary lambda terms is standard. However, the monadic components require explanation.

Note that state-layer monadic configurations must reduce to a $\mathtt{return}_{\mathsf{M}}$ in their respective layers. This amounts to requiring that monadic terms have normal forms. That is to say, this reflects a natural requirement of termination—namely, that monadic-reduction performed with regards to a stateful computation results in a value.

Stores feature prominently in the monadic part of our development. It is only natural, then, that we require that stores satisfy two reducibility conditions relative to their corresponding monadic types. We require that terms contained in stores must be values, and that those terms must be in the reducibility sets of their underlying types. These requirements correspond to `store_all_values` and the fixpoint definition `Rsto` in Figure 9. As the name suggests, `Rsto` is simply a reducibility requirement for stores.

We embed the coinductive predicate `along_react` inside the fixpoint definition of **R** as a condition on terms typed in the reactive layer. An added difficulty is that `along_react` needs to be defined lexically prior to the definition of **R**. As such, `along_react` mentions neither **R**, nor `Rsto`. Instead, we must use partial application—i.e., $(\mathbf{R}\,\tau)$. These technicalities notwithstanding, the structure of the constructors for `along_react` is fairly straightforward – involving routine reasoning for coinduction.

4.4 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 does not alter stores where no writes are allowed (Theorem 4.13); and moreover, that monadic-reduction does not reveal any changes to stores relative to monads with effect labels where only reads are allowed (Theorem 4.15). To that end,

```
CoInductive along_react : (store → Prop) → (tm → Prop) → (tm → Prop) → (tm → Prop) → configuration → Prop :=
  | along_return : ∀ (PS:store → Prop) (PI PO PR:tm → Prop) τi τo SM t Σ t' Σ′,
                   (t,Σ) ⇝* (return(ReT τi τo SM) t',Σ′) → value t' → PR t' → PS Σ′ → along_react PS PI PO PR (t,Σ)
  | along_pause  : ∀ (PS : store → Prop) (PI PO PR : tm → Prop) τi τo SM τ t Σ t1 Σ′ vl vr Σ″,
                   (t,Σ)  ⇝* (pause(ReT τi τo SM τ) t1,Σ′)                              →
                   (t1,Σ′) ⇝* (returnSM ⟨vl,vr⟩,Σ″)                                      →
                   value t1 ∧ value vl ∧ value vr                                        →
                   PS Σ′ ∧ PS Σ″ ∧ PO vl                                                 →
                   (∀ t1, PI t2 → halts (vr t2))                                         →
                   (∀ t2 Σ″, PI t2 → PS Σ″ → along_react PS PI PO PR ((vr t2),Σ″)) →
                   along_react PS PI PO PR (t,Σ).
```

Fig. 10. The Coinductive Predicate along_react

$$\frac{}{\mathsf{nil} \overset{\mathsf{Id}(\cancel{W})}{=} \mathsf{nil}}\ (\textsc{swnw-id}) \qquad \frac{\Sigma \overset{S(\cancel{W})}{=} \Sigma'}{\Sigma \overset{\mathsf{ReT}\ \tau\ \tau'\ S(\cancel{W})}{=} \Sigma'}\ (\textsc{swmw-re})$$

$$\frac{\Sigma \overset{S(\cancel{W})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau\langle\rangle\, S(\cancel{W})}{=} (s :: \Sigma')}\ (\textsc{swnw-n}) \qquad \frac{\Sigma \overset{S(\cancel{W})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau\langle R\rangle\, S(\cancel{W})}{=} (s :: \Sigma')}\ (\textsc{swnw-r})$$

$$\frac{\Sigma \overset{S(\cancel{W})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau\langle W\rangle\, S(\cancel{W})}{=} (s' :: \Sigma')}\ (\textsc{swnw-w})$$

$$\frac{\Sigma \overset{S(\cancel{W})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau\langle RW\rangle\, S(\cancel{W})}{=} (s' :: \Sigma')}\ (\textsc{swnw-rw})$$

Fig. 11. The 'same where no write' relation.

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

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

$$\frac{\Sigma \overset{S(\mathsf{R})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau\langle W\rangle\, S(\mathsf{R})}{=} (s' :: \Sigma')}\ (\textsc{swr-w})$$

$$\frac{\Sigma \overset{S(\mathsf{R})}{=} \Sigma'}{(s :: \Sigma) \overset{\mathsf{StT}\ \tau\langle RW\rangle\, S(\mathsf{R})}{=} (s :: \Sigma')}\ (\textsc{swr-rw})$$

Fig. 12. The 'same where read' relation.

$$\frac{}{\langle \mathsf{nil}, \mathsf{nil}\rangle\ wc\ \langle \mathsf{nil}, \mathsf{nil}\rangle}\ (\textsc{wc-id})$$

$$\frac{\langle \Sigma_1, \Sigma_2\rangle\ wc\ \langle \Sigma_1', \Sigma_2'\rangle}{\langle s_1 :: \Sigma_1, s_2 :: \Sigma_2\rangle wc\ \langle s_1 :: \Sigma_1', s_2 :: \Sigma_2'\rangle}\ (\textsc{wc-unchanged})$$

$$\frac{\langle \Sigma_1, \Sigma_2\rangle\ wc\ \langle \Sigma_1', \Sigma_2'\rangle}{\langle s_1 :: \Sigma_1, s_2 :: \Sigma_2\rangle wc\ \langle s :: \Sigma_1', s :: \Sigma_2'\rangle}\ (\textsc{wc-changed})$$

Fig. 13. The write consistency relation.

we make use of three relations: "same where no writes", "same where read", and "write consistency", written $\overset{M(\cancel{W})}{=}$, $\overset{M(\mathsf{R})}{=}$, and *wc*—and defined in Figure 11, Figure 12, and Figure 13—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. Lemma 4.12 contains properties used to prove Theorem 4.13.

Lemma 4.12. *For all stores* $\Sigma, \Sigma^{'}, \Sigma^{''}$*, and monads* $\mathsf{M}, \mathsf{M}^{'}$

(1) *If* Σ *matches* M*, then* $\Sigma \overset{\mathsf{M}(\cancel{W})}{=} \Sigma$.
(2) *if* $\Sigma \overset{\mathsf{M}(\cancel{W})}{=} \Sigma^{'}$ *and* $\Sigma^{'} \overset{\mathsf{M}(\cancel{W})}{=} \Sigma^{''}$*, then* $\Sigma \overset{\mathsf{M}(\cancel{W})}{=} \Sigma^{''}$.
(3) *If* M *is less permissive than* $\mathsf{M}^{'}$ *and* $\Sigma \overset{\mathsf{M}(\cancel{W})}{=} \Sigma^{'}$*, then* $\Sigma \overset{\mathsf{M}^{'}(\cancel{W})}{=} \Sigma^{'}$.

Theorem 4.13 (No Forbidden Updates). *If* $\langle t, \Sigma \rangle \triangleright M\ \tau$*, then* $\langle t, \Sigma \rangle \rightsquigarrow \langle t', \Sigma' \rangle$ *implies* $\Sigma \overset{M(\cancel{W})}{=} \Sigma'$.

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. Lemma 4.14 contains properties used to prove Theorem 4.15.

Lemma 4.14. *For all stores* $\Sigma, \Sigma^{'}, \Sigma^{''}$*, and monads* $\mathsf{M}, \mathsf{M}^{'}$

(1) *If* Σ *is the same length as* $\Sigma^{'}$*, then* $\langle \Sigma, \Sigma^{'} \rangle$ *wc* $\langle \Sigma, \Sigma^{'} \rangle$.
(2) *If* $\Sigma \overset{\mathsf{M}(R)}{=} \Sigma^{'}$*, then* Σ *is the same length as* $\Sigma^{'}$.
(3) *If* $\langle s :: \Sigma \rangle \overset{\mathsf{StT}\ \tau \ell\ S(R)}{=} \langle s' :: \Sigma' \rangle$*, then* $\langle s :: \Sigma \rangle \overset{\mathsf{S}(R)}{=} \langle s' :: \Sigma' \rangle$.
(4) *If* $\ell \leq \mathsf{R}$ *and* $\langle s :: \Sigma \rangle \overset{\mathsf{StT}\ \tau \ell\ S(R)}{=} \langle s' :: \Sigma' \rangle$*, then* $s = s'$.
(5) *If* M *is less permissive than* $\mathsf{M}^{'}$ *and* $\Sigma \overset{\mathsf{M}^{'}(R)}{=} \Sigma^{'}$*, then* $\Sigma \overset{\mathsf{M}(R)}{=} \Sigma^{'}$.

Theorem 4.15 (No Forbidden Reads). *Suppose* $\Sigma_1 \overset{M(R)}{=} \Sigma_2$ *and that* $\langle t, \Sigma_1 \rangle \triangleright M\ \tau$ *and* $\langle t, \Sigma_2 \rangle \triangleright M\ \tau$*. 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).

The pre-stores, as stated in Theorem 4.15, must satisfy the 'same where read' relation. The type system, because of its effect labels and their ordering, restricts admissible alterations to terms and when such changes can be read from stores. This is particularly useful for equational reasoning involving security properties such as noninterference as shown in Figure 14. Theorem 4.15 "says": if Σ_1 and Σ_2 are in the 'same where read' relation, then executing term t in Σ_1 and Σ_2 produces both equal resulting terms, t_1 and t_2, resp., as well as write-consistent pre- and post-stores.

5 TYPE-DIRECTED EQUATIONAL LOGIC FOR REWIRE CALCULUS

The rules provided in Figure 14 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->>=) are the "lifting laws" of Liang [43]. Rules (Put-Put), (Put-Get), and (Get-Get), specify the interaction of stateful operations and are drawn from previous work [30]. The ≤ relation on state monads is defined in Figure 5.

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 14. 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.

$$\frac{t = t' : \tau \in \Gamma}{\Gamma \vdash t = t' : \tau}\ (\textsc{Axiom}) \qquad \frac{\Gamma \vdash t = t' : \tau}{\gamma, \Gamma \vdash t = t' : \tau}\ (\textsc{Weakening})$$

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

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

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

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

$$\frac{\Gamma \vdash \mathsf{return}_{\mathsf{M}}\ t : \mathsf{M}\,\tau}{\Gamma \vdash \mathsf{lift}_{\mathsf{M}'}(\mathsf{return}_{\mathsf{M}}\ t) = \mathsf{return}_{\mathsf{M}'}\ t : \mathsf{M}'\,\tau}\ (\textsc{Lift-Return})$$

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

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

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

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

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

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

Fig. 14. Type-directed Equational Logic for RWC

6 CONCLUSIONS AND FUTURE WORK

This article has presented a mechanized formal semantics for the functional hardware description language ReWire and, as such, provides a foundation for high assurance hardware design and implementation. The semantics presented here is of the small-step operational variety, which is, at first blush, somewhat surprising. ReWire is a computational λ-calculus in the sense of Moggi [48], and, therefore, possesses a "built-in" denotational semantics based in categorical language semantics [47] which has been discussed elsewhere [56]. But, generally speaking, small-step operational semantics are more readily mechanized in a theorem prover like Coq and this was a primary motivation for pursuing an operational approach. All of the definitions and theorems in this paper have been checked with the Coq proof checker (v8.5) and are downloadable [59].

Synchronous hardware is generally assumed to be non-terminating and that motivates the use of ReWire's core abstraction—potentially infinite resumption-monadic computations—for modeling hardware [58]. Formalizing resumptions in Coq involved technical challenges that required some ingenuity to overcome; these challenges and our approach to overcoming them were discussed in detail in Section 4. To the authors' best knowledge, the coinductive style of defining logical relations in Coq is apparently an innovation that may be of use to other researchers in formal methods and interactive theorem proving.

ReWire inherits its purity (i.e., freedom from side effects) from Haskell, and purity, in turn, made the task of formally specifying ReWire relatively straightforward. Were ReWire embedded in an impure functional language (e.g., OCaml or Scala[5]), its resulting semantic specification would have necessarily been more complicated in order to account for the host language's side effects. Any model of synchronous hardware will be complex—but, that being said, the purity of ReWire contributed to simplifying its formalized semantics.

The ReWire methodology differs fundamentally from the type-based approach to secure hardware (e.g., that of Caisson [42], Sapper [41], and SecVerilog [83]) 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 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 [61]. 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 [30, 58] is supported by an effects type system based on the marriage of effects and monads [80]. 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 type-checking approach of Caisson and SecVerilog and traditional hardware verification with theorem provers [45]. 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. Delite—a compiler framework for parallel embedded domain-specific languages (EDSLs) targeted to produce hardware—exhibits what its creators call "the three P's" [39]: productivity, performance and portability. Our previous work [27, 28, 58] demonstrates that ReWire possesses what "the three P's" [39] and 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 [27, 58].

The CompCert [40] 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 [78]) 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] D. Andrews. 2015. Will the Future Success of Reconfigurable Computing Require a Paradigm Shift in Our Research Community's Thinking? Keynote address, Applied Reconfigurable Computing. (2015). http://hthreads.csce.uark.edu/mediawiki/images/d/d8/Arc-presentation.pdf.

[2] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. Pierce, R. Pollack, and A. Tolmach. 2014. A Verified Information-flow Architecture. In *POPL*. 165–178.

[3] C. Baaij and J. Kuper. 2014. Using Rewriting to Synthesize Functional Languages to Digital Circuits. In *Trends in Fun. Prog. (LNCS)*, Vol. 8322. 17–33.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *DAC*. 1216–1225.

[5] Homepages: https://ocaml.org and https://www.scala-lang.org, respectively.

[5] D. Bacon, R. Rabbah, and S. Shukla. 2013. FPGA Programming for the Masses. *Queue* 11, 2, Article 40 (Feb. 2013), 13 pages.

[6] L. Baugh, N. Neelakantam, and C. Zilles. 2008. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08).* 115–126.

[7] R. Bird and P. Wadler. 1988. *Introduction to Functional Programming.* Prentice Hall.

[8] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. 1998. Lava: Hardware design in Haskell. In *3rd ICFP.* 174–184.

[9] T. Braibant and A. Chlipala. 2013. Formal Verification of Hardware Synthesis. In *CAV.* 213–228.

[10] G. Cabodi and M. Murciano. 2006. BDD-Based Hardware Verification. In *6th Inter. Conf. on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'06).* 78–107.

[11] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind. 2017. Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

[12] K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279.

[13] D. Cock, G. Klein, and T. Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *TPHOLs.* 167–182.

[14] Coq [n. d.]. The Coq Proof Assistant. ([n. d.]). https://coq.inria.fr.

[15] T. Coquand. 1994. *Infinite objects in type theory.* Springer Berlin Heidelberg, Berlin, Heidelberg, 62–78. https://doi.org/10.1007/3-540-58085-9_72

[16] K. Crary, A. Kliger, and F. Pfenning. 2005. A monadic analysis of information flow security with mutable state. *JFP* 15, 2 (March 2005), 249–291.

[17] C. Doczkal and J. Schwinghammer. 2009. Formalizing a Strong Normalization Proof for Moggi's Computational Metalanguage: A Case Study in Isabelle/HOL-nominal. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP '09).* ACM, New York, NY, USA, 57–63. https://doi.org/10.1145/1577824.1577834

[18] Jean H. Gallier. 1990. On Girard's "Candidates de Reducibilite". In *Logic and Computer Science.* Academic Press, 123–204.

[19] P. Gammie. 2013. Synchronous Digital Circuits As Functional Programs. *ACM Comput. Surv.* 46, 2, Article 21 (Nov. 2013), 27 pages. https://doi.org/10.1145/2543581.2543588

[20] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. 2014. Hardware system synthesis from Domain-Specific Languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL).* 1–8.

[21] D. Ghica and A. Jung. 2016. Categorical semantics of digital circuits. In *FMCAD.*

[22] E. Giménez. 1995. *Codifying guarded definitions with recursive schemes.* Springer Berlin Heidelberg, Berlin, Heidelberg, 39–59. https://doi.org/10.1007/3-540-60579-7_3

[23] J.-Y. Girard, Y. Lafont, and P. Taylor. 1989. *Proofs and types.* Vol. 7. Cambridge University Press Cambridge.

[24] J.A. Goguen and J. Meseguer. 1984. Unwinding and Inference Control. In *IEEE Symp. on Security and Privacy.* 75–86.

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

[26] M. Gordon. 1995. The semantic challenge of Verilog HDL. In *Logic in Computer Science, 1995. LICS '95. Proceedings., Tenth Annual IEEE Symposium on.* 136–145.

[27] I. Graves, W. Harrison, A. Procter, and G. Allwein. 2015. Provably Correct Development of Reconfigurable Hardware Designs via Equational Reasoning. In *IEEE Inter. Conf. on Field-Programmable Technology (ICFPT).* 160–171.

[28] I. Graves, A. Procter, W. Harrison, M. Becchi, and G. Allwein. 2015. Hardware Synthesis from Functional Embedded Domain-Specific Languages: A Case Study in Regular Expression Compilation. In *Applied Reconfigurable Computing (LNCS)*, Vol. 9040. 41–52.

[29] W. Harrison. 2006. The Essence of Multitasking. In *Algebraic Methodology and Software Technology.* 158–172.

[30] W. Harrison and J. Hook. 2009. Achieving information flow security through monadic control of effects. *JCS* 17 (Oct 2009), 599–653. Issue 5.

[31] W. Harrison, A. Procter, and G. Allwein. 2016. Model-driven Design & Synthesis of the SHA-256 Cryptographic Hash Function in ReWire. In *Proceedings of the 27th International Symposium on Rapid System Prototyping (RSP).* 1–7.

[32] W. Harrison, A. Procter, I. Graves, M. Becchi, and G. Allwein. 2016. A Programming Model for Reconfigurable Computing Based in Functional Concurrency. In *11th Inter. Symp. on Reconfigurable Communication-centric Systems-on-Chip.*

[33] Bluespec Homepage. 2017. http://bluespec.com. (July 2017).

[34] B. Huffman. 2012. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs.* Ph.D. Dissertation. Portland State University.

[35] T. Huffmire, C. Irvine, T. Nguyen, T. Levin, R. Kastner, and T. Sherwood. 2010. *Handbook of FPGA Design Security.* Springer.

[36] T. Huffmire, S. Prasad, T. Sherwood, and R. Kastner. 2006. Policy-Driven Memory Protection for Reconfigurable Hardware. In *ESORICS*. LNCS, Vol. 4189. 461–478.

[37] T. Huffmire, T. Sherwood, R. Kastner, and T. Levin. 2008. Enforcing memory policy specifications in reconfigurable hardware. *Computers & Security* 27, 5–6 (2008), 197 – 215.

[38] C. Kloos and P. Breuer (Eds.). 1995. *Formal Semantics for VHDL.* Kluwer Academic Publishers.

[39] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. 2011. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro* 31, 5 (Sept. 2011), 42–53.

[40] X. Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115.

[41] X. Li, V. Kashyap, J. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. Chong. 2014. Sapper: A Language for Hardware-level Security Policy Enforcement. In *ASPLOS*.

[42] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. Chong, T. Sherwood, and B. Hardekopf. 2011. Caisson: a hardware description language for secure information flow. In *PLDI*. 109–120.

[43] S. Liang, P. Hudak, and M. Jones. 1995. Monad Transformers and Modular Interpreters. In *POPL*. 333–343.

[44] A. Megacz. 2012. Hardware Design with Generalized Arrows. In *Proceedings of the 23rd International Conference on Implementation and Application of Functional Languages (IFL'11).* Springer-Verlag, Berlin, Heidelberg, 164–180. https://doi.org/10.1007/978-3-642-34407-7_11

[45] T. Melham. 1993. *Higher Order Logic and Hardware Verification.* Cambridge Tracts in Theoretical Computer Science, Vol. 31. Cambridge University Press.

[46] J. Mitchell. 1996. *Foundations for Programming Languages.* MIT Press Cambridge.

[47] E. Moggi. 1990. *An Abstract View of Programming Languages.* Technical Report ECS-LFCS-90-113. Department of Computer Science, Edinburgh University.

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

[49] A. Myers. 2017. personal communication. (March 2017).

[50] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *ICFP*. 229–240.

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

[52] R. S. Nikhil and Arvind. 2009. What is Bluespec? *SIGDA Newsl.* 39, 1 (Jan. 2009), 1–1. https://doi.org/10.1145/1862876.1862877

[53] S. Ouchani, O. A. Mohamed, and M. Debbabi. 2013. A formal verification framework for Bluespec System Verilog. In *Proceedings of the 2013 Forum on specification and Design Languages (FDL).* 1–7.

[54] S. Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries, the Revised Report.* Cambridge University Press. 272 pages.

[55] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey. 2015. *Software Foundations.* Electronic textbook.

[56] A. Procter. 2014. *Semantics-Driven Design and Implementation of High-Assurance Hardware.* Ph.D. Dissertation. University of Missouri, 2014. Department of Computer Science.

[57] A. Procter, W. Harrison, I. Graves, M. Becchi, and G. Allwein. 2015. Semantics Driven Hardware Design, Implementation, and Verification with ReWire. In *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES).*

[58] A. Procter, W. Harrison, I. Graves, M. Becchi, and G. Allwein. 2017. A Principled Approach to Secure Multi-core Processor Design with ReWire. *ACM TECS* 16, 2, Article 33 (Feb. 2017), 33:1–33:25 pages.

[59] Code repository for MEMOCODE. 2017. https://goo.gl/FYf6xU. (July 2017).

[60] D. Richards and D. Lester. 2011. A monadic approach to automated reasoning for Bluespec SystemVerilog. *Innovations in Systems and Software Engineering* 7, 2 (18 Mar 2011), 85. https://doi.org/10.1007/s11334-011-0149-0

[61] A. Sabelfeld and A. Myers. 2003. Language-based Information-flow Security. *IEEE Journ. on Sel. Areas in Commun.* 21, 1 (Jan. 2003).

[62] I. Sander and A. 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.

[63] I. Sander and A. Jantsch. 2008. Modelling Adaptive Systems in ForSyDe. *Electronic Notes in Theoretical Computer Science* 200, 2 (2008), 39–54. https://doi.org/10.1016/j.entcs.2008.02.011

[64] D. Sangiorgi. 2009. On the Origins of Bisimulation and Coinduction. *ACM Trans. Program. Lang. Syst.* 31, 4 (May 2009), 15:1–15:41.

[65] L. Schröder and T. Mossakowski. 2009. HasCasl: Integrated higher-order specification and program development. *Theoretical Computer Science* 410, 12 (2009), 1217 – 1260.

[66] M. Sheeran. 1984. muFP, a Language for VLSI Design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 104–112. https://doi.org/10.1145/800055.802026

[67] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404

[68] W. Swierstra. 2009. A Hoare Logic for the State Monad. In *TPHOLs*. 440–451.

[69] W. W. Tait. 1967. Intensional interpretations of functionals of finite type I. *The journal of symbolic logic* 32, 2 (1967), 198–212.

[70] W. W. Tait. 1975. A Realizability Interpretation of the Theory of Species. In *Logic Colloquium (Lectures Notes in Mathematics)*, R. Parikh (Ed.), Vol. 453. Springer-Verlag, Boston, 240–251.

[71] M. Tehranipoor and C. Wang. 2011. *Introduction to Hardware Security and Trust*. Springer Publishing Company, Incorporated.

[72] M. Tiwari, Xun Li, H.M.G. Wassel, F.T. Chong, and T. Sherwood. 2009. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. 493–504.

[73] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. 2011. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *Proceedings of the 38th Annual ISCA*. 189–200.

[74] M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/1508244.1508258

[75] M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/1508244.1508258

[76] S. M. Trimberger and J. J. Moore. 2014. FPGA Security: Motivations, Features, and Applications. *Proc. IEEE* 102, 8 (Aug 2014), 1248–1265.

[77] D. Volpano, C. Irvine, and G. Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2-3 (Jan. 1996), 167–187. http://dl.acm.org/citation.cfm?id=353629.353648

[78] VST [n. d.]. Verified Software Toolchain. http://vst.cs.princeton.edu. ([n. d.]).

[79] A. Procter W. Harrison and G. Allwein. 2012. The Confinement Problem in the Presence of Faults. In *ICFEM*. 182–197.

[80] P. Wadler. 1998. The Marriage of Effects and Monads. In *ICFP*. 63–74.

[81] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 225–240. http://dl.acm.org/citation.cfm?id=1855741.1855757

[82] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards. 2015. Hardware Synthesis from a Recursive Functional Language. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis (CODES '15)*. IEEE Press, Piscataway, NJ, USA, 83–93. http://dl.acm.org/citation.cfm?id=2830840.2830850

[83] D. Zhang, Y. Wang, G. E. Suh, and A. Myers. 2014. *A Hardware Design Language for Efficient Control of Timing Channels*. Technical Report 2014-04-10. Dept. of Computer Science, Cornell University. Extended version of the authors' ASPLOS15 paper.

Received December 2017