Formalized High Level Synthesis with Applications to Cryptographic Hardware*

William Harrison[0000−0002−3760−3556], Ian Blumenfeld[0000−0003−2080−9790], Eric Bond, Chris Hathhorn[0000−0002−6277−3987], Paul Li[0009−0000−0789−7410], May Torrence, and Jared Ziegler

Two Six Technologies, Inc.
901 N. Stuart Road, Arlington VA 22203. USA.

Abstract. Verification of hardware-based cryptographic accelerators connects a low-level RTL implementation to the abstract algorithm itself; generally, the more optimized for performance an accelerator is, the more challenging its verification. This paper introduces a verification methodology, *model validation*, that uses a formalized high-level synthesis language (FHLS) as an intermediary between algorithm specification and hardware implementation. The foundation of our approach to model validation is a mechanized denotational semantics for the ReWire HLS language. Model validation proves the faithfulness of FHLS models to the RTL implementation and we summarize a model validation case study for a suite of pipelined Barrett multipliers.

Keywords: Programming languages and models · Verifying cryptographic systems · Automated theorem proving.

1 Introduction

This paper presents the mechanized semantics for the functional high-level synthesis (HLS) language ReWire [48, 53], where ReWire is an embedded DSL in Haskell for expressing synchronous hardware designs. This semantics is the cornerstone of a hardware verification methodology called *model validation* that we also introduce with the verification case study of a family of cryptographic accelerators for fully homomorphic encryption. With model validation, ReWire plays a dual role as a language for both formal modeling and implementation.

Model validation (Fig. 1) establishes that a Verilog design produces the same results as a verified correct ReWire model. The first path (model; embed; verify) creates a ReWire model, embeds it in a theorem prover via ReWire's formalized semantics, and verifies its functional correctness. The second path (model ; validate) validates

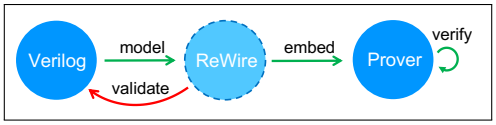


Fig. 1: Model Validation Methodology.

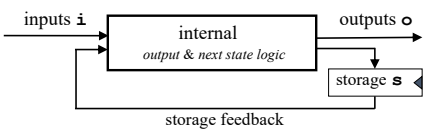
* This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

the fidelity of the ReWire model to the Verilog design by establishing functional equivalence using the model-checking capabilities in YoSys [59].

Synchronous circuitry never terminates and, consequently, neither do ReWire programs. ReWire syntax and semantics are structured by reactive resumption monads over state (RRS), where computations in RRS monads [18, 34, 43, 45, 57] resemble potentially infinite sequences of stateful actions. Non-terminating computation can be challenging to mechanize with a theorem prover and, for the ReWire semantics, this challenge is overcome by an alternative representation of RRS monads using infinite streams. This stream-based RRS representation allows an embedding of ReWire directly into any prover with a stream library—we provide example embeddings of the semantics in Isabelle, Coq, and Agda [12]. The semantics resembles a Reynolds-style definitional interpreter [52], although our semantics targets theorem prover object languages rather than a general-purpose functional programming language as Reynolds' classic paper did. The shallow embedding uses effect labels [41] to distinguish between the termination behavior of ReWire terms and to selectively pick the appropriate denotations.

This focus of this paper is primarily on the `embed` arrow in Fig. 1 and we leave a broader discussion of model validation and its uses for follow-on publications. The remainder of this section introduces background on ReWire. Section 2 presents the formalization of ReWire as a typed λ-calculus and the embedding of this semantics in three theorem proving systems: Isabelle, Coq, and Agda. It is with the Isabelle embedding that we perform the formal verification of the family of pipelined Barrett multipliers in Section 3. Section 3 describes the BMM case study at a high-level due to lack of space. Section 4 reviews related work and Section 5 summarizes our results and outlines future directions for this research.

ReWire is a domain-specific language (DSL) embedded in Haskell for expressing, implementing, and verifying hardware designs. All ReWire programs are Haskell programs (but not necessarily vice versa). We assume of necessity that the reader is familiar with functional languages and especially with the use of monads to model effects in functional programming (see Appendix A for an overview). We first illustrate ReWire syntax and semantics in terms of two simple examples: Mealy machines and carry-save adders.



(a) Mealy Machine

```
internal  :: i → ST s o
internal i = ...
onecycle  :: i → Re i s o i
onecycle i = lift (internal i) >>= λo. signal o
mealy     :: i → Re i s o ()
mealy i    = onecycle i >>= mealy
```

(b) ReWire Mealy Design Template

Fig. 2: Mealy Machines (a); Corresponding Mealy Template in ReWire (b).

Mealy machines (Fig. 2a) are a common mental model for designers of sequential circuitry [26,35]. Given current values of the input (`i`), internal storage

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

(s), and output (o), the internal combinational logic of the Mealy machine computes the storage and output values for the next clock cycle. Fig. 2b presents a ReWire template encoding the Mealy machine. The type constructors, `Re i s o` and `ST s`, refer, respectively, to a reactive resumption monad over state and to the state monad. The type variables `i`, `s`, and `o` in `Re i s o` correspond directly to the Mealy machine's input, storage, and output types. Monads like `Re i s o` and `ST s` possess their respective monadic unit (`return`) and bind (`>>=`) operators (that are typically overloaded in both Haskell and ReWire). Operations in `ST s` read and write storage typed in `s`. The `Re` operation `lift` injects a stateful computation into `Re` and `signal` performs synchronous input-output.

It is possible to describe what `mealy` does intuitively before presenting any formal semantics (although readers experienced with monadic semantics may find Fig. 4 useful at this point). Calls to `onecycle` describe exactly one clock cycle of circuit execution, while calls to `mealy` describe an entire circuit computation itself. The internal action of a cycle, `lift (internal i)`, in combination with the current internal storage (of type `s`), updates that storage, and computes the next output `o`. The `signal` operator sends its argument to the output ports and, then, returns the next input. Producing a signal, (`signal o`), sends the computed output to the output port, and signifies the completion of a clock cycle; `mealy` then continues, ad infinitum. `ST` (resp., `Re`) operations will ultimately be compiled into combinational (resp., sequential) circuitry by the ReWire compiler.

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c = ( ((a & b) | (a & c) | (b & c)) << 1 , a ⊕ b ⊕ c )
data Ans a = DC | Val a   — resp., "don't care" and "valid"

csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8)
csa (a, b, c) = signal (f a b c) >>= csa

scsa :: (W8, W8, W8) → Re (W8, W8, W8) (W8, W8) (W8, W8) ()
scsa abc = save abc >>= λcs. signal cs >>= scsa
  where
    thread :: (W8, W8) → ST (W8, W8) (W8, W8)
    thread cs = set cs >> get
    save :: (W8 , W8 , W8) → Re (W8 , W8 , W8) (W8, W8) (W8, W8) (W8, W8)
    save (a , b , c) = lift (thread (f a b c))

pcsa :: W8 → Re W8 () (Ans (W8, W8)) ()
pcsa a = signal DC >>= λb. signal DC >>= λc. signal (Val cs) >>= pcsa
                where cs = f a b c

bad :: i → Re i i o ()   — Haskell, not ReWire; not signal-productive
bad i = lift (set i) >>= bad
```

Fig. 3: ReWire source code for Carry-Save Adder Functions. The operators &, |, and ⊕ are bitwise and, inclusive or, and exclusive or. Operator << is shift-left.

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

A carry-save adder (CSA) is a function which takes in three n-bit words `a`, `b`, and `c`, and computes two n-bit words `s` and `c'`, such that $\texttt{a} + \texttt{b} + \texttt{c} = \texttt{s} + \texttt{c}'$. Fig. 3 presents three ReWire functions for CSA circuits for $n = 8$. The function `f` defines the carry-save operation, so that, for example, `f 40 25 20` $= (48, 37)$, representing `W8` words as integers for readability. The answer `Ans` data type indicates whether an output is valid. Function `csa` accepts inputs `a`, `b`, and `c` on each clock cycle, computes their carry-save sum, and sends that sum to the outport port before starting again. The behavior of `scsa` is the same as `csa`, but `scsa` also stores the result in a local store of type $(\texttt{W8}, \texttt{W8})$—this difference is reflected in the types of `csa` and `scsa` in Fig. 3. Function `pcsa` is pipelined, accepting inputs on successive clock cycles and computing the carry-save sum when the third input, `c`, is available. While it waits, `DC` is `signal`ed, and, once all three arguments are available, `Val` of the carry-save sum is `signal`ed.

The `bad` function in Fig. 3 is not valid ReWire because it is not *signal-productive*—i.e., there is no output-producing call to `signal`. Signal-productivity means that ReWire programs regularly produce outputs analogously to how synchronous circuits (e.g., `mealy` in Fig. 2a) produce outputs on every clock signal. Signal-productivity is enforced by the type system below in Section 2 (e.g., so that `bad` does not type check).

The ReWire compiler can translate functions like `mealy`, `csa`, `scsa`, and `pcsa` into synthesizable VHDL or Verilog (as shorthand, we call such functions *devices*). But not every Haskell function with codomain `Re i s o a` is a device—there are three main provisos arising from the nature of synchronous hardware—and none of these provisos is enforced by the Haskell type system. The first proviso limits recursion in devices to tail recursion, because tail recursion only requires a fixed memory footprint. Arbitrary recursive Haskell functions may require a stack and heap and such dynamic allocation is anathema to hardware. The second proviso requires that devices never terminate—i.e., just like a synchronous circuit, they should (in principle) never terminate on any inputs. The third proviso is that they be signal-productive—the Haskell function `bad` in Fig. 3 is not signal-productive and, hence, is not a ReWire device. The effect type system described in Section 2 enforces each these requirements so that $\texttt{Re}^{\infty}$ ($\texttt{Re}^{+}$) is the type for devices (resp., signal-productive, terminating terms).

A conventional formulation of `Re` appears in Fig. 4. In ReWire, `Re` is constructed using Haskell monad transformers, but rather than introducing that notational overhead here, we define `Re` directly in Fig. 4. The functor part of `Re` is written in a categorical style followed by the definitions of its unit (`return`) and bind (`>>=`). Additional structure includes `lift` (which lifts a stateful computation into `Re`) and `signal` (which sends `o` to the "output port"). We include these definitions for reference and to make the article self-contained.

2 Formalizing ReWire

The ReWire formalization is a conventionally structured denotational semantics of the form, $[\![-]\!] : (\Gamma \vdash \texttt{t}) \to \texttt{Env}\,\Gamma \to \ulcorner \texttt{t} \urcorner$, mapping a well-typed term and

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

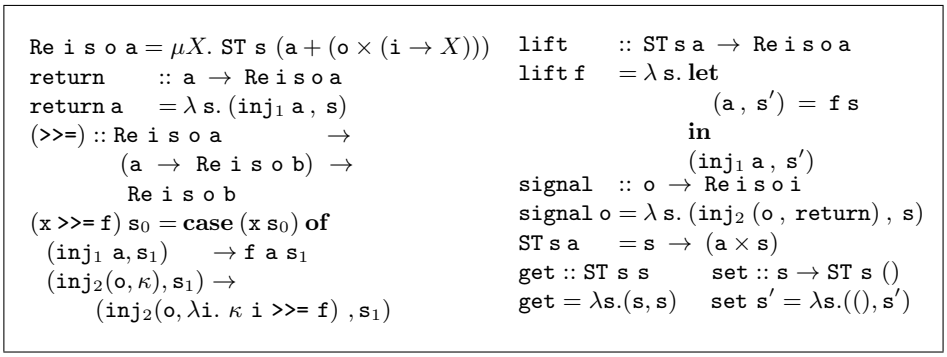


Fig. 4: *Reactive Resumption Monads over State.* `Re` is a synchronous concurrency monad allowing expression of both terminating and non-terminating threads; it constitutes a core part of ReWire's syntax and semantics. The codebase includes a Haskell rendering of this semantics [12].

suitable environment into a domain of values. We first present the term and type syntax of the formalized ReWire effect calculus and then the mechanization of RRS monads. RRS monads originated in the denotational semantics of concurrent and parallel languages [18, 34, 43, 45, 57]; much of the challenge of formalizing ReWire originates in representing them in a theorem prover.

We use the term *denotational* advisedly for our semantics, because the term may evoke expectations in some readers of some explicit form of CPO semantics. The ReWire semantics takes the form of, to borrow a term from Reynolds [52], a *definitional interpreter*—i.e., an embedding of a source language into a conventional functional programming language. Here, however, the embedding maps a typed syntax for ReWire into the object language of a theorem prover (specifically Isabelle, Agda, and Coq). The domain semantics displayed in Fig. 6a is based on infinite streams of snapshots and this enabled the straightforward definitional embedding of ReWire into Isabelle, Coq, and Agda, because each of these provers possesses a stream library. This obviated the need for a deep embedding of the denotational semantics in the manner of, for example, Huffman et al. [24, 25] or Schröder [54]. We present the Agda formalization because Agda's syntax is simpler to read than either that of Coq or Isabelle [12], and within that code, several syntactic simplifications have been made to improve readability (e.g., removing certain quantifiers or implicitly-passed variables, etc.).

ReWire is a computational λ-calculus (in the sense of Moggi [37]) with monadic constructs corresponding to the `Re` and `ST` monads from Fig. 4. The type language in Fig. 5a includes effect labels indicating the termination and productivity behavior of expressible programs. The intrinsically-typed term syntax encodes typing rules in the constructors. The type language contains base types specific to hardware: `bit` and the standard logic vector type constructor (`slv`) that takes a natural number representing bit vector size. We elide operations on low-level data types in Fig. 5a because they are not remarkable.

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

```
data Ty : Set where
  nat  : Ty
  bool : Ty
  unit : Ty
  bit  : Ty
  slv  : ℕ → Ty
  _⇒_  : Ty → Ty → Ty
  _⊗_  : Ty → Ty → Ty
  _⊕_  : Ty → Ty → Ty
  ST   : Ty → Ty → Ty
  Re⁰  : Ty → Ty →
         Ty → Ty → Ty
  Re⁺  : Ty → Ty →
         Ty → Ty → Ty
  Re^∞ : Ty → Ty →
         Ty → Ty
```

Effects

```
p, q ∈ {0 , + , ∞}
0 ⊔ 0  = 0   + ⊔ + = +
0 ⊔ +  = +   + ⊔ 0 = +
```

```
data _⊢_ : Cxt Ty → Ty → Set where
  var      : a ∈ Γ → Γ ⊢ a
  lam      : a :: Γ ⊢ b → Γ ⊢ (a ⇒ b)
  app      : Γ ⊢ a ⇒ b → Γ ⊢ a → Γ ⊢ b
                 ... elided ...
  return^ST : Γ ⊢ a ⇒ ST s a
  _>>=^ST_ : Γ ⊢ ST s a →
                 Γ ⊢ a ⇒ ST s b →
                 Γ ⊢ ST s b
  get      : Γ ⊢ ST s s
  set      : Γ ⊢ s ⇒ ST s unit
  lift^r   : Γ ⊢ ST s a ⇒ Re⁰ i s o a
  _>>=^pq_ : Γ ⊢ Re^p i s o a ⇒
                 (a ⇒ Re^q i s o b) ⇒
                 Re^(p⊔q) i s o b
  return^r : Γ ⊢ a ⇒ Re⁰ i s o a
  signal^r : Γ ⊢ o ⇒ Re⁺ i s o i
  loop     : Γ ⊢ (a ⇒ Re⁺ i s o a) ⇒
                 (a ⇒ Re^∞ i s o)
```

(a) Type and Effect Syntax (left) and intrinsically-typed term syntax (right).

```
⌜_⌝            : Ty → Set
⌜nat⌝          = ℕ
⌜bool⌝         = Bool
⌜unit⌝         = ⊤
⌜bit⌝          = Bool
⌜slv n⌝        = Vec Bool n
⌜t₁ ⇒ t₂⌝      = ⌜t₁⌝ → ⌜t₂⌝
⌜t₁ ⊗ t₂⌝      = ⌜t₁⌝ × ⌜t₂⌝
⌜t₁ ⊕ t₂⌝      = ⌜t₁⌝ ⊎ ⌜t₂⌝
⌜ST s a⌝       = State ⌜s⌝ ⌜a⌝
⌜Re⁰ i s o a⌝ =
  DomRe⁰ ⌜i⌝ ⌜s⌝ ⌜o⌝ ⌜a⌝
⌜Re⁺ i s o a⌝ =
  DomRe⁺ ⌜i⌝ ⌜s⌝ ⌜o⌝ ⌜a⌝
⌜Re^∞ i s o⌝ =
  DomRe^∞ ⌜i⌝ ⌜s⌝ ⌜o⌝
```

```
⟦ − ⟧             : (Γ ⊢ t) → Env Γ → ⌜t⌝
⟦ var x ⟧ ρ       = lookup∈ ρ x
⟦ lam f ⟧ ρ       = λv. ⟦ f ⟧ (v ◁ ρ)
⟦ app f e ⟧ ρ     = (⟦ f ⟧ ρ) (⟦ e ⟧ ρ)
        ⋮
⟦ return^ST ⟧ ρ   = λv. SM(λs. (v , s))
⟦ e >>=^ST f ⟧ ρ  = (⟦ e ⟧ ρ) <>=^00 (⟦ f ⟧ ρ)
⟦ get ⟧ ρ         = SM (λs. (s , s))
⟦ set ⟧ ρ         = λs. SM (λ_. (() , s))
⟦ lift^r ⟧ ρ      = λφ. φ
⟦ return^r ⟧ ρ    = λv. SM(λs. (v , s))
⟦ e >>=^pq f ⟧ ρ  = (⟦ e ⟧ ρ) <>=^pq (⟦ f ⟧ ρ)
⟦ signal ⟧ ρ      = λo. λ(_ , s , _). λis.
                      let
                        i  = shd is
                        is′ = stl is
                      in
                        (i , s , o) ▷ [(i , is′)]
⟦ loop f ⟧ ρ      = iterRe (⟦ f ⟧ ρ)
```

(b) Denotational Semantics for the ReWire Calculus

Fig. 5: ReWire as an Effect Calculus.

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

The syntax is parameterized by productivity labels, `0`, $+$, and ∞, which are ordered linearly so that `p` $\sqcup$ `q` returns the maximum of labels `p` and `q`. Terms of `0`-productivity are created with $\texttt{lift}^{\texttt{r}}$ and $\texttt{return}^{\texttt{r}}$ or binds of `0`-productive computations. Such computations correspond to computations by combinational circuitry between clock cycles. Terms of $+$-productivity are created with `signal` or binds, `x >>= f`, in which at least one of `x` or `f` is a $+$-productive computations. Computations typed in $\texttt{Re}^{+}$ correspond to signal-productive, terminating computations spanning at least one clock cycle. One could define $\texttt{>>=}^{pq}$ for cases in which p and/or q is ∞, but we have not done so here. In Haskell, for example, `x >>= f` is identical to `x` when `x` is non-terminating; such terms are not of use in expressing hardware designs in ReWire. Terms of ∞-productivity—i.e., what we previously called devices—may be only created with the recursion-binder `loop`. To represent the `mealy` program from Fig. 2b in the ReWire Calculus, one would refactor its definition with `loop` so that $\texttt{mealy} : \texttt{i} \rightarrow \texttt{Re}^{\infty}\ \texttt{i}\ \texttt{s}\ \texttt{o}$ and $\texttt{mealy} = \texttt{loop}\ \texttt{onecycle}$. Refactoring with a recursion operator is a common syntactic change of representation in denotational semantics.

Fig. 5b defines the denotational semantics of the ReWire calculus. It is worth remarking on its structure and organization now, but detailed discussion is deferred until the end of this section. The domain semantics ($\ulcorner - \urcorner$) maps each type `Ty` into a corresponding Agda `Set`. For the RRS monadic type constructors, there are corresponding constructions indexed by effect labels and these are defined in the next section. Most of the cases in the semantics of terms ($[\![-]\!]$) are similarly not remarkable except in the monadic cases. Corresponding to syntactic binds (i.e., $\texttt{>>=}^{pq}$) are semantic binds (i.e., $\texttt{<>=}^{pq}$) and corresponding to recursive syntactic operator (`loop`) is the semantic recursive operator (`iterRe`).

Reynolds et al. [53] formulated a small-step, operational semantics for ReWire in Coq. A deep embedding formalizing ReWire's denotational semantics [47] in terms of mechanized domain theory (e.g., Huffman [24, 25], Benton et al. [5], or Schröder [54]) is possible as well. However, both the deep embedding and the small-step operational approaches seemed too unwieldy at the scale of our case studies. Recent work [22] introduced the Device Calculus, a λ-calculus with types and operations for constructing Mealy machines and our semantics extends the Device Calculus semantics to RRS monads.

Fig. 6a presents the semantics for reactive resumption monads over state in which the productivity-labelled constructors are expressed in terms of "snapshots" of the form $(\texttt{i}, \texttt{s}, \texttt{o})$. `State s` is the familiar state monad over `s`. A ($\texttt{Writer}^{+}\ \texttt{s}\ \texttt{a}$) is a list-like structure for which the constructor $\triangleright$ corresponds to list cons—intuitively, it is a non-empty list that ends in an `a`-value—and is used to model ReWire terms typed in $\texttt{Re}^{+}$. It is used to represent terminating *signal-productive* hardware computations—i.e., those that operate over multiple clock cycles, produce snapshots and terminate. A hardware computation typed in $\texttt{Re}^{\infty}$ corresponds to sequential circuitry. The intuition is that, given the current snapshot of a circuit (Fig. 2a) and a stream of all its future inputs, the result is a stream of all snapshots (i.e., a $\texttt{Stream}\ (\texttt{i} \times \texttt{s} \times \texttt{o})$).

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

```
data State (s : Set) (a : Set) : Set where
  SM : (s → (a×s)) → State s a

data Writer+ (w : Set) (a : Set) : Set where
  _▷[_] : w → a → Writer+ w a
  _▷_   : w → Writer+ w a → Writer+ w a

DomRe0 i s o a = State s a
DomRe+ i s o a = (i×s×o) → Stream i → Writer+ (i×s×o) (a × Stream i)
DomRe∞ i s o   = (i×s×o) → Stream i → Stream (i×s×o)
```

(a) Domain Semantics

```
_<>=00_  : ST s a → (a → ST s b) → ST s b
_<>=0+_  : ST s a → (a → DomRe+ i s o b) → DomRe+ i so b
_<>=++_ : DomRe+ i s o a → (a → DomRe+ i s o b) → DomRe+ i s o b
iterRe   : (a → DomRe+ i s o a) → (a → DomRe∞ i s o a)
```

(b) Type Declarations of Effect-labeled Bind & Co-Recursion Operators

Fig. 6: Domain Semantics & Semantic Operators.

Signal-productive computations (i.e., those corresponding to terms of type `Re+ i s o a`) are represented in the domain `DomRe+ ⌜i⌝ ⌜s⌝ ⌜o⌝ ⌜a⌝`. The intuition underlying this structure is that, given an initial snapshot (`i` , `s` , `o`) and a stream of inputs in `i`, signal-productive computations will express a finite, non-zero number of additional snapshots, represented in `Writer+ (i × s × o) a`. The intuition underlying `DomRe∞ ⌜i⌝ ⌜s⌝ ⌜o⌝ ⌜a⌝` is similar, except that it produces a stream expressing the entire circuit as a "transcript" of snapshots. The intuition underlying a value in `DomRe0 ⌜i⌝ ⌜s⌝ ⌜o⌝ ⌜a⌝` is simple—it produces no snapshots because it represents computation that occurs between clock cycles; hence it is simply a state monad computation.

The type declarations for effect-labeled bind operators are shown in Fig. 6b. The monad laws for these were verified in Coq [12]. We chose to verify these laws in Coq and, although this choice was somewhat arbitrary, it does however illustrate the utility of Reynold-style definitional shallow embedding of the ReWire formalization. The Coq syntax below is different from the Agda syntax we have adopted throughout; e.g., `bindRePP` stands for (`>>=++`), etc. A typical theorem, showing the associativity of (`>>=++`), is below:

```
Theorem AssocPP {i s o a b c} : forall (x : RePlus i s o a),
 forall (f : a -> RePlus i s o b), forall (g : b -> RePlus i s o c),
   bindRePP x (fun va => bindRePP (f va) g) = bindRePP (bindRePP x f) g.
```

Fig. 6b presents the type declaration of the corecursion operator, `iterRe`. ReWire devices typically take the form of mutually recursive co-equations and such co-equations may be encoded in the ReWire calculus using a standard

approach from denotational semantics. Two ReWire co-equations (left) are represented in the calculus by (`iterRe f`), where `f` is defined as (right):

$$\begin{array}{ll} \mathtt{f_i} \quad :: \mathtt{a_i} \rightarrow \mathtt{Re\ i\ s\ o}\ () & \mathtt{f} : (\mathtt{a_1} \oplus \mathtt{a_2}) \rightarrow \mathtt{Re}^{+}\ \mathtt{i\ s\ o} \\ \mathtt{f_1}\ \mathtt{a_1} = \mathtt{x_1}\ \mathtt{>>=}\ \mathtt{f_2} & \mathtt{f}\ (\mathtt{inl}\ \mathtt{a_1}) = \mathtt{x_1}\ \mathtt{>>=}^{+0}\ (\mathtt{return}^{\mathtt{r}} \circ \mathtt{inr}) \\ \mathtt{f_2}\ \mathtt{a_2} = \mathtt{x_2}\ \mathtt{>>=}\ \mathtt{f_1} & \mathtt{f}\ (\mathtt{inr}\ \mathtt{a_2}) = \mathtt{x_2}\ \mathtt{>>=}^{+0}\ (\mathtt{return}^{\mathtt{r}} \circ \mathtt{inl}) \end{array}$$

Fig. 5b presents the mechanized denotational semantics for ReWire. It closely resembles the Device Calculus semantics referred to previously [22], except for the monadic fragment of the calculus, which is represented by the constructions of Fig. 6a. The state monadic operators ($\mathtt{return}^{\mathtt{ST}}$, $\mathtt{>>=}^{\mathtt{ST}}$, `get`, and `set`) have an unremarkable semantics. Lifting and unit (respectively, $\mathtt{lift}^{\mathtt{r}}$ and $\mathtt{return}^{\mathtt{r}}$) are treated as state monad computations as one would expect from the type semantics in Fig. 6a. Lifting is the identity function and the denotation for $\mathtt{return}^{\mathtt{r}}$ is identical to that of $\mathtt{return}^{\mathtt{ST}}$. The productivity-labelled bind is mapped to the appropriate operator from Fig. 6b. The denotation of `signal` computes a snapshot (`i` , `s` , `o`) based on the current internal state (`s`), the head of the input stream (`i`), and the output argument it has been passed (`o`), returning the next input and the remaining stream of inputs. The semantics of `loop` applies `iterRe` to the denotation of `f`.

3 Case Study: Cryptographic Hardware Verification

We performed the model validation process on a substantial case study: a family of pipelined Barrett modular multipliers (BMM) that are based on hardware algorithms published by Zhang et al. [62]. The formal methods team was provided with Verilog designs created by hand by a team of hardware engineers and it was our task was to formally verify the correctness of these designs. The designs in question were highly optimized using a variety of techniques (e.g., specialized encodings for compression/decompression) to enhance area and time performance of the synthesized circuits. The technical focus heretofore has been on the embed arrow from Fig. 1. This section summarizes the BMM case study (i.e., the verify arrow in Fig. 1) and we provide sufficient information to understand the its essentials, although the presentation is necessarily at a high-level due to space limitations. A complete description is left for future publications.

It is important to note that the Verilog designs for BMM were not designed with formal verification in mind. Model validation is a hybrid approach mixing interactive theorem-proving with user-guided, but otherwise, fully automated equivalence checking. We developed this approach, in part, because we were concerned that a fully-automated approach would not scale up to the large size of several of the designs. All of the relevant materials to this case study are available [12]; these include Verilog designs for the multipliers, the Isabelle proof scripts that specify and verify the hardware designs, as well as the semantics for ReWire formalized in Isabelle, Coq, and Agda.

***BMM Case Study* (model).** Creating a ReWire model of the BMM Verilog design constitutes the model phases of the model validation process illustrated in

```
module BMM (CLK, A_IN, B_IN, M_IN
            , mu_IN, km3_IN, Z_OUT);

 parameter N     = 128;
 parameter LOG_N = 7;
 input                CLK;
 input  [N-1     : 0] A_IN, B_IN, M_IN;
 input  [N+2     : 0] mu_IN;
 input  [LOG_N-1 : 0] km3_IN;
 output [N-1   : 0] Z_OUT;
 reg [2*N-1 : 0] stage0_XY_reg; // stage 0 reg
 reg [N+2   : 0] stage0_mu_reg;
 reg [N-1   : 0] stage0_M_reg;
 reg [LOG_N-1:0] stage0_km3_reg;
 reg [N     : 0] stage1_XY_reg; // stage 1 reg
 reg [N-1   : 0] stage1_q_reg;
 reg [N-1   : 0] stage1_M_reg;
 reg [N     : 0] stage2_XY_reg; // stage 2 reg
            ...
```

(a) Input Verilog for BMM

```
type Inp = ( BV(N)       -- A_IN
           , BV(N)       -- B_IN
           , BV(N)       -- M_IN
           , BV(N + 3)   -- mu_IN
           , BV(LOG_N) ) -- km3_IN
type Out = BV(N)         -- Z_OUT

bmm :: Inp → Re Inp Reg Out ()
bmm i = do lift (internal i)
           i' ← signal (obs reg)
           bmm i'
    where
        internal :: Inp → ST Reg Out
        internal i = do r ← get
                        put (trans i r)
                        returnST (obs r)
        trans :: Inp → Reg → Reg
        trans i r = ...
        obs :: Reg → Out
        obs r     = ...
```

(b) Corresponding ReWire Mimic

Fig. 7: Case Study: Modeling Hardware Designs in ReWire. The `bmm` function (b) is an instance of the ReWire's Mealy pattern that mimics the original hardware design (a). Haskell's `do` notation is syntactic sugar for `>>=`.

Fig. 1. The task required formally verifying instances of this input RTL for word sizes: W = 64, 128 , 256, 512, and 1024. This section illustrates this process using relevant parts of the BMM case study. An excerpt of the input BMM Verilog code is presented in Fig. 7a. The top-level input and output declarations are displayed (not all register declarations are included for reasons of space).

The Verilog I/O port declarations that are captured as ReWire tuple types, `Inp` and `Out`, in Fig. 7b. The Verilog register declarations are encoded as the ReWire tuple type, `Reg`, although it does not appear in the figure. The ReWire compiler unfolds boolean vector types to built-in ReWire types (e.g., for `N` = 128, `BV(N)` becomes the built-in ReWire word type `W128`).

One notable difference between the Verilog input ports and the ReWire type `Inp` in Fig. 7 is the absence of a clock type in the latter. This reflects the implicit timing inherent in the `Re` monad. Fig. 7b excerpts the ReWire formal model that mimics the input Verilog BMM design—this is a ReWire function, `bmm`, that has type `Inp → Re Inp Reg Out`. The ReWire function `bmm` is an instance of the Mealy design pattern from Fig. 2b. In our experience, most of the effort in the model phase of model validation derives from specifying the input, storage, and output types (e.g., `Inp`, `Reg`, and `Out`)and, also, from the formulation of the `internal` function that represents the combinational output and next-state logic. Developing the ReWire model was, for the case study presented here, entirely by hand, although we believe that future work can automate (at least parts of) the process (see Section 5 for further discussion).

***BMM Case Study* (embed).** The final part of the embed arrow in Fig. 1 for this case study is the semantic translation of the ReWire model into the logic of the Isabelle theorem prover. (Some liberties have been taken with Is-

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

abelle syntax for readability.) The semantic foundation expressed in Figures 5 and 6 was developed as a theory file in Isabelle. This development was along the lines of Reynold's notion of definitional interpreters [52] as remarked upon in the previous section—i.e., because Isabelle possesses a stream library, the definitional embedding of the ReWire semantics was straightforward. For example, the semantic domain $\texttt{DomRe}^{\infty}$ `i s o` is formulated in Isabelle in Fig. 8. Given this semantic foundation formulated as an Isabelle theory, a translation into this theory based on the denotational semantics from Fig. 5b was written in Haskell. This translation, in most respects, simply transliterates ReWire abstract syntax into the constructions of the Isabelle semantic theory, making use of the built-in monadic syntax in Isabelle/HOL. Fig. 8 presents the Isabelle translation of the ReWire mimic of the original BMM design (from Fig. 7b). Note the structural similarity between `body` and `onecycle` from Fig. 2b. Note also that `body` is typed in the Isabelle version of $\texttt{DomRe}^{+}$ from Fig. 6a. The translator analyses recursive definitions (e.g., the original `bmm` from Fig. 7b) and reformulates them using `iterRe`, but, otherwise, the translations of ReWire definitions in Fig. 8 are unremarkable. The use of Oxford brackets emphasizes that this Isabelle declaration defines the denotational semantics of `bmm` from Fig. 7b.

```
type_synonym ('i,'s,'o) DomRe_INF =
           "('i × 's × 'o) ⇒ 'i stream ⇒ (('i × 's × 'o) stream)"
fun body :: "Inp ⇒ (Inp, Reg, Out, Inp) Dom_Re_Plus" where
   "body (i) = retdo { reg ← liftR get;
                       liftR (set (trans i reg)); signalR (obs reg) }"
definition ⟦bmm⟧ :: "Inp ⇒ (Inp, Reg, Out) Dom_Re_INF"
  where "⟦bmm⟧ i = iterRe body (i)"
```

Fig. 8: Embedding of `bmm` from Fig. 7b in Isabelle.

***BMM Case Study* (verify).** This section presents the verify phase of the model validation process illustrated in Fig. 1. The `compute_bmm` function in Fig. 9 defines the calling convention for the `bmm` ReWire device. In the figure, the initial values, `i0`, `s0`, and `o0`, are tuples of zeros, represented as bit vectors of appropriate sizes (e.g., `o0` is just `W128`). The function applies ⟦`bmm`⟧ to the appropriate inputs thereby producing a stream of snapshots. The computed `bmm` value is the output of the fifth such snapshot (calculated with projection π_3, stream take `stake`, and the list indexing operation "`!`"). The correctness theorem `embedding_eq` in Fig. 9 is expressed in Isabelle as an equation relating the results computed by the `compute_bmm` Isabelle embedding (lhs) to the value computed by the high-level algorithm, `barrett_fws_word` (rhs).

***BMM Case Study* (validate).** This section overviews the validate phase of the model validation process illustrated in Fig. 1 as applied to the BMM case study. The successful proof of the correctness theorem `embedding_eq` in Isabelle verifies the functional correctness of the ReWire representation of the BMM target design. This alone provides a strong assurance story, but there remains

a question as to the accuracy of the hand translation of Verilog BMM design into ReWire model—what evidence is there that the ReWire model faithfully represents the input design? Model validation goes further and demonstrates the soundness of the model through the use of model checking technology.

```
fun compute_bmm :: "128 word ⇒ 128 word ⇒ 128 word ⇒ 131 word ⇒ 7 word ⇒ 128 word"
 where
  "compute_bmm a b m mu km3 =
    π3 (stake 5 (⟦bmm⟧ (a,b,m,mu,km3) (i0,s0,o0)  (repeat (a,b,m,mu,km3))) ! 4)"

theorem embedding_eq : "compute_bmm a b m mu km3 = barrett_fws_word a b m mu km3"
```

Fig. 9: Formal Specification of `bmm`.

The *yosys* (Yosys Open SYnthesis Suite) toolchain [59] supports the synthesis of Verilog (and, through an extension, VHDL) designs, providing an array of options for transformation, optimization, and model checking. In particular for our use case, *yosys* integrates the ABC system [9] for sequential logic synthesis and formal verification. Here, we use *yosys* to carry out an equivalence check between two circuits: those synthesized from the input Verilog BMM design and the Verilog output by compiling the verified ReWire model.

The ReWire compiler provides a Verilog backend and we can thus perform an apples-to-apples comparison of the two Verilog circuits using *yosys*. Because the ReWire model mimics the modular and algorithmic structure of the hand-written circuit, *yosys* can quickly identify common substructures in support of automatic equivalence verification of the two circuits. Even with the high degree of similarity between the two circuits, some of the more complex equivalence checks proved challenging for the automated tooling. To break down the problem further, we applied *compositional verification*, in which subcomponents are verified individually and those results are used to verify higher-level components. After we verify equivalence for a submodule, we instruct *yosys* to treat references to that submodule by both the implementation and ReWire specification as a blackbox library. "Blackboxing" modules can streamline equivalence checking.

The *yosys* scripts we used may be found in the codebase [12]. Our initial experimentation focused on purely combinational circuits, provable using the *yosys* `equiv_simple` command. This worked "out of the box" for a number of submodules. However, much of the target design consists of sequential circuits, which require additional configuration to manage timing and state. In this case, with the `equiv_induct` command, *yosys* proves such circuits equivalent by temporal induction over clock cycles.

4 Related Work

We coined the term *model validation* because of its similarities to translation validation [17, 40, 44, 46]. Translation validation begins with a given source program and compiler and, then, establishes the correctness relation between the

source and its implementation (i.e., the compiled source program). Translation validation establishes the correctness of individual compiler translations rather than verifying the whole compiler itself. Model validation starts from a given implementation (i.e., the HDL circuit design) and high-level correctness criteria (e.g., an algorithm given in pseudocode) and, then, establishes the equivalence of the two to a ReWire formal model that mimics the circuit design. The (model ; validate) path in model validation proceeds in the "opposite direction" from translation validation. Translation validation for HLS has been applied before (e.g., Kundu [29] and Ramanathan et al. [49,50]), but model validation is novel to the best knowledge of the authors.

High-level synthesis (HLS) adapts software high-level languages to hardware development. The motivation to do so has been to bring software engineering virtues—e.g., modularity, comprehensibility, reusability, etc.—to the whole hardware development process [2] but also more recently to translate software formal methods into a hardware context [7,14,53]. Herklotz and Wickerson [23] and Du et al. [13] make compelling arguments for applying software formal methods to HLS languages and compilers as a means of bringing a level of maturity and reliability to HLS that justifies its use in critical systems. Formal methods applied to *software* compilers have been explored for at least five decades now [38] and the state of the art is at a high-level of sophistication [30].

Gordon outlined the challenges of semantic specification of hardware definition languages [19], focusing specifically on Verilog, although his analysis applies equally to VHDL. There have been previous attempts to formalize VHDL as well [28, 58] that have succeeded only on small parts of the language. One way of coping with the lack of formal semantics for commodity HDLs is to identify a formalizable subset of the language in question. Gordon [20], Zhu et al. [63], Meredith et al. [36], Khan [27], and Lööw and Myreen [32] do so for Verilog. Another approach creates a new hardware language and compiler with formalization as a specific requirement (e.g., Kami [11], Bluespec [7], and CHERI [42]). HLS generally seeks to adapt software languages to hardware—ReWire, being a DSL embedded in Haskell, is in this camp.

The original motivation for high-level synthesis was to promote software-like development to hardware design by introducing software-like abstractions and methodologies. In particular, functional language approaches to high-level synthesis have a long pedigree, including muFP [55], Cλash [15], ForSyDe, Lava [6], Kiwi [56], and Chisel [4]. There is a growing awareness of the utility of language-based approaches (including HLS) for hardware formal methods (e.g., a sample of very recent publications [3, 7, 8, 21, 22, 31, 32, 42, 53] can be found in the references). This language-based approach has been particularly successful in formal development of instruction set architectures [3, 42, 51].

There has been work formalizing monads with theorem provers as a basis for verifying functional programs [1,10,16,33,39]. Simple monads (e.g., Haskell's `Maybe`) can be transliterated into a theorem prover, but more complex monads—e.g., RRS monads—require more care [24,25,53,54]; their mechanization here is, by comparison, a shallow embedding. Effect labels in the ReWire calculus type

system were essential because they allow fine-grained distinctions with respect to signal-productivity and non-termination to be made in the construction of terms that, in turn, determine the appropriate denotation domain.

5 Summary, Conclusions, and Future Work

The research described here was performed as part of a project to develop formally verified hardware accelerators to improve upon the existing algorithmic gains to fully-homomorphic encryption (FHE). ReWire's role is to bridge the gap between the hardware design and algorithm by establishing 1) the equivalence of the algorithm to the model and 2) the equivalence of the model to the circuit design. Equivalence between the algorithm and the ReWire model is verified with a ReWire semantics formalized in the Isabelle theorem prover. Equivalence between the ReWire model and the input circuit design is established by producing binary circuits from each (using commodity synthesis tools and the ReWire compiler) and applying an automated binary equivalence checker.

Model validation addresses the following kind of scenario. A team of hardware engineers produces a circuit design C in a commodity HDL (e.g., VHDL or Verilog) to implement an algorithm A (written in informal, imperative style pseudocode) in hardware and then a formal methods team is given the task of evaluating whether C implements A correctly. There is significant distance between the notions of computation underlying A (i.e., store-passing in some form) and C (i.e., clocked, synchronous parallelism) and so formally relating the two is non-trivial and requires care. We have shown how a formalized HLS language like ReWire can bridge this gap to reduce this conceptual distance.

The first path of model validation—the composite arrow (model; embed; verify) in Fig. 1—is, in some respects, a conventional hardware verification flow with a theorem prover: a formal model is abstracted from an HDL design, encoded in the prover logic, and then properties of that model are verified. The interposed formalized HLS language may provide some benefits with respect to proof engineering via libraries of theorems that may be reused later. We have developed such libraries of theorems and tactics over the course of this project that will be shared as open source. The second path of the model validation process—the composite arrow (model ; validate)—speaks to the fidelity of the formal model itself to the input circuit design. Establishing the fidelity of a formal model to the object it models addresses a broad issue in formal methods research that can be difficult to explore: how can we gauge the accuracy of a formal model itself?

The class of high-level algorithms of which the BMM case study is a member are generally informally specified as C-style pseudocode (see, for instance, Zhang et al. [62]). One approach for future work would be to develop a formalized domain-specific language for this class of high-level algorithms that can be lifted automatically into ReWire. This would accelerate the model validation process as it would automate the otherwise time-consuming, by-hand model phase. Such a language-based approach would support, among other things, a correct-by-construction approach to hardware development based in program transforma-

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

tion. Another potential accelerator applies recent work by Zeng et al. [60, 61] that seeks to automatically generate update functions of type $\mathtt{i} \to \mathtt{s} \to (\mathtt{o} \times \mathtt{s})$ from Verilog designs. Automatic recovery of such update functions would go a long way towards automating the model phase of model validation.

We have successfully applied the model validation methodology to several substantial case studies, including the BMM case study from Section 3 and another on a 4096-bit iterative Montgomery modular multiplier (MMM) that we will describe in future work. Why develop a new methodology at all? Several members of the formal methods team have extensive experience with Cryptol, for example, and we did experiment with it. For example, we specified some of the basic encoder components from the MMM in Cryptol, but the automated equivalence check of these against the relevant components failed to terminate after several days. It seemed unlikely, then, that this fully automated approach would scale up to a 4096-bit multiplier. One of the key reasons for our success in these case studies is the extensive automation available in Isabelle—that motivated our choice of Isabelle over Coq. ReWire is open source and the success of the (model ; validate) path in Fig. 1 relied on our ability to make customizations to its Verilog code generator in support of Yosys equivalence checking.

	Fmax (GHz)			Area (μm^2)		
Width	ReWire	Original	Δ%	ReWire	Original	Δ%
64	1.588	2.127	+25%	13399	12126	+10%
128	1.357	2.134	+36%	42970	41650	+3%
256	1.229	1.952	+37%	150463	157214	-4%
512	1.074	1.789	+40%	554612	578506	-4%
1024	0.954	1.473	+35%	2109037	2106714	+0.1%

Table 1: Performance Comparison: ReWire vs. Handwritten Barrett Multipliers.

Comparing the performance of the compiled ReWire models in Section 3 against those of the original Verilog designs was in some respects surprising to us. Table 1 displays performance numbers (maximum clock frequency and area) for the case study for each word size of pipelined Barrett multipliers. The columns labeled "Original" are those for the original Verilog design created by hand and those labeled "ReWire" are for the mimic designs created as formal models. While the maximum clock frequency numbers for the ReWire models are between 25%-40% slower than the Original designs, the area of the circuits is roughly equivalent and, in some cases, slightly better than those produced for the handwritten designs. Future work will explore the optimization of the ReWire compiler to bring these performance characteristics into line with hand-written Verilog and VHDL designs as much as possible.

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

References

1. Affeldt, R., Nowak, D., Saikawa, T.: A hierarchy of monadic effects for program verification using equational reasoning. In: Hutton, G. (ed.) Mathematics of Program Construction. pp. 226–254 (2019)
2. Andrews, D.: Will the future success of reconfigurable computing require a paradigm shift in our research community's thinking? (Apr 2015), keynote address, Applied Reconfigurable Computing
3. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: The State of Sail. In: SpISA 2019: Workshop on Instruction Set Architecture Specification (September 2019)
4. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanovic, K.: Chisel: constructing hardware in a scala embedded language. In: DAC. pp. 1216–1225 (2012)
5. Benton, N., Kennedy, A., Varming, C.: Some domain theory and denotational semantics in Coq. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. pp. 115–130. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
6. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in haskell. ACM SIGPLAN Notices **34** (05 2001)
7. Bourgeat, T., Pit-Claudel, C., Chlipala, A., Arvind: The essence of Bluespec: A core language for rule-based hardware design. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 243–257. PLDI 2020 (2020)
8. Bourke, T., Brun, L., Pouzet, M.: Mechanized semantics and verified compilation for a dataflow synchronous language with reset. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019)
9. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Computer Aided Verification. pp. 24–40 (2010)
10. Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., Weirich, S.: Ready, set, verify! applying hs-to-coq to real-world haskell code (experience report). Proc. ACM Program. Lang. (jul 2018)
11. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., Arvind: Kami: a platform for high-level parametric hardware specification and its modular verification. PACMPL **1**, 24:1–24:30 (2017)
12. Model Validation Codebase. Available from `https://www.dropbox.com/s/r59xg34qzh0arri/codebase_paper4262.tar.gz?dl=0` (Dec 2022)
13. Du, Z., Herklotz, Y., Ramanathan, N., Wickerson, J.: Fuzzing high-level synthesis tools. In: The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. p. 148. FPGA '21, Association for Computing Machinery, New York, NY, USA (2021)
14. Flor, J.P.P., Swierstra, W., Sijsling, Y.: Π-Ware: Hardware Description and Verification in Agda. In: Proc. TYPES (2015)
15. Gerards, M., Baaij, C., Kuper, J., Kooijman, M.: Higher-order abstraction in hardware descriptions with CλaSH. In: Proceedings of the 2011 14th EUROMICRO Conference on Digital System Design. pp. 495–502. DSD '11, IEEE Computer Society, Washington, DC, USA (2011). `https://doi.org/10.1109/DSD.2011.69`, `http://dx.doi.org/10.1109/DSD.2011.69`

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

16. Gibbons, J., Hinze, R.: Just do it: Simple monadic equational reasoning. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. p. 2–14. ICFP '11 (2011)
17. Goldberg, B., Zuck, L., Barrett, C.: Into the loops: Practical issues in translation validation for optimizing compilers. Electronic Notes in Theoretical Computer Science **132**(1), 53–71 (2005), proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004)
18. Goncharov, S., Schröder, L.: A coinductive calculus for asynchronous side-effecting processes. In: Proc. of the 18th International Conf. on Fundamentals of Computation Theory. pp. 276–287 (2011)
19. Gordon, M.J.C.: The semantic challenge of Verilog HDL. Proc. of 10th Annual IEEE LICS pp. 136–145 (1995)
20. Gordon, M.J.C.: Relating Event and Trace Semantics of Hardware Description Languages. The Computer Journal **45**(1), 27–36 (01 2002)
21. Harrison, W.L., Allwein, G.: Verifiable security templates for hardware. In: Proceedings of the Design, Automation, and Test Europe (DATE) Conference (2020)
22. Harrison, W.L., Hathhorn, C., Allwein, G.: A mechanized semantic metalanguage for high level synthesis. In: 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021) (2021)
23. Herklotz, Y., Wickerson, J.: High-level synthesis tools should be proven correct. In: Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) (2021)
24. Huffman, B.: HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs. Ph.D. thesis, Portland State University (2012)
25. Huffman, B.: Formal verification of monad transformers. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 15–16. ICFP '12 (2012)
26. Katz, R.H.: Contemporary Logic Design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. (2000)
27. Khan, W., Tiu, A., Sanan, D.: Veriformal: An executable formal model of a hardware description language. In: Roychoudhury, A., Liu, Y. (eds.) A Systems Approach to Cyber Security: Proceedings of the 2nd Singapore Cyber-Security R&D Conference (SG-CRC 2017), pp. 19–36. IOS Press (2017)
28. Kloos, C., Breuer, P. (eds.): Formal Semantics for VHDL. Kluwer Academic Publishers (1995)
29. Kundu, S., Lerner, S., Gupta, R.K.: Translation Validation of High-Level Synthesis, pp. 97–121. Springer New York, New York, NY (2011). https://doi.org/10.1007/978-1-4419-9359-5_7, https://doi.org/10.1007/978-1-4419-9359-5_7
30. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (Jul 2009)
31. Lööw, A.: Lutsig: A verified verilog compiler for verified circuit development. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 46–60. CPP 2021, Association for Computing Machinery, New York, NY, USA (2021)
32. Lööw, A., Myreen, M.O.: A Proof-Producing Translator for Verilog Development in HOL. In: 2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE). pp. 99–108 (2019)
33. Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hriţcu, C., Rivas, E., Tanter, E.: Dijkstra monads for all. Proc. ACM Program. Lang. **3**(ICFP) (jul 2019)

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

34. Marlow, S., Brandy, L., Coens, J., Purdy, J.: There is no fork: An abstraction for efficient, concurrent, and concise data access. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 325–337. ICFP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628144, http://doi.acm.org/10.1145/2628136.2628144
35. Mealy, G.H.: A method for synthesizing sequential circuits. The Bell System Technical Journal **34**(5), 1045–1079 (Sep 1955)
36. Meredith, P., Katelman, M., Meseguer, J., Roşu, G.: A formal executable semantics of Verilog. In: Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010). pp. 179–188 (July 2010)
37. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55–92 (July 1991)
38. Morris, F.L.: Advice on structuring compilers and proving them correct. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 144–152. POPL '73, Association for Computing Machinery, New York, NY, USA (1973)
39. Mu, S.C.: Calculating a backtracking algorithm: an exercise in monadic program derivation. Tech. Rep. TR-IIS-19-003, Institute of Information Science, Academia Sinica (June 2019)
40. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming language design and implementation. pp. 83–94 (2000)
41. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (2010)
42. Nienhuis, K., Joannou, A., Bauereiss, T., Fox, A., Roe, M., Campbell, B., Naylor, M., Norton, R.M., Moore, S.W., Neumann, P.G., Stark, I., Watson, R.N.M., Sewell, P.: Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In: 2020 IEEE Symposium on Security and Privacy. pp. 1003–1020 (2020)
43. Papaspyrou, N.S.: A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. In: Proceedings of the 3rd Panhellenic Logic Symposium (2001), expanded version available as a tech. report from the author by request.
44. Perez, I., Goodloe, A.: Copilot 3. Tech. Rep. 20200003164, National Aeronautics and Space Administration (NASA) (04 2020)
45. Piróg, M., Gibbons, J.: The coinductive resumption monad. Electronic Notes in Theoretical Computer Science **308**, 273 – 288 (2014). https://doi.org/http://dx.doi.org/10.1016/j.entcs.2014.10.015, http://www.sciencedirect.com/science/article/pii/S1571066114000826, proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXX)
46. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 151–166. Springer (1998)
47. Procter, A.: Semantics-Driven Design and Implementation of High-Assurance Hardware. Ph.D. thesis, University of Missouri (2014)
48. Procter, A., Harrison, W., Graves, I., Becchi, M., Allwein, G.: A principled approach to secure multi-core processor design with ReWire. ACM TECS **16**(2), 33:1–33:25 (Jan 2017)
49. Ramanathan, N., Constantinides, G.A., Wickerson, J.: Concurrency-aware thread scheduling for high-level synthesis. In: 2018 IEEE 26th Annual International Sym-

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

posium on Field-Programmable Custom Computing Machines (FCCM). pp. 101–108 (2018). https://doi.org/10.1109/FCCM.2018.00025
50. Ramanathan, N., Fleming, S.T., Wickerson, J., Constantinides, G.A.: Hardware synthesis of weakly consistent C concurrency. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. p. 169–178. FPGA '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3020078.3021733, https://doi.org/10.1145/3020078.3021733
51. Reid, A.: Trustworthy specifications of ARM® v8-a and v8-m system level architecture. In: 2016 Formal Methods in Computer-Aided Design (FMCAD). pp. 161–168 (Oct 2016). https://doi.org/10.1109/FMCAD.2016.7886675
52. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference - Volume 2. p. 717–740. ACM '72 (1972)
53. Reynolds, T.N., Procter, A., Harrison, W., Allwein, G.: The mechanized marriage of effects and monads with applications to high-assurance hardware. ACM TECS **18**(1), 6:1–6:26 (Jan 2019)
54. Schröder, L.: Bootstrapping inductive and coinductive types in hascasl. Logical Methods in Computer Science **4**(4:17), 1–27 (2008)
55. Sheeran, M.: muFP, a language for VLSI design. In: LISP and Functional Programming. pp. 104–112 (1984)
56. Singh, S., Greaves, D.J.: Kiwi: Synthesis of FPGA circuits from parallel programs. In: 2008 16th International Symposium on Field-Programmable Custom Computing Machines. pp. 3–12 (2008). https://doi.org/10.1109/FCCM.2008.46
57. Swierstra, W., Altenkirch, T.: Beauty in the beast. In: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop. pp. 25–36. Haskell '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1291201.1291206, http://doi.acm.org/10.1145/1291201.1291206
58. Umamageswaran, K., Pandey, S.L., Wilsey, P.A.: Formal Semantics and Proof Techniques for Optimizing VHDL Models. Kluwer Academic Publishers, USA (1999)
59. Wolf, C.: Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/
60. Zeng, Y., Gupta, A., Malik, S.: Automatic generation of architecture-level models from RTL designs for processors and accelerators. In: Design, Automation & Test in Europe (DATE). pp. 460–465 (March 2022). https://doi.org/10.23919/DATE54114.2022.9774527
61. Zeng, Y., Huang, B.Y., Zhang, H., Gupta, A., Malik, S.: Generating architecture-level abstractions from RTL designs for processors and accelerators part i: Determining architectural state variables. In: 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD). pp. 1–9 (2021)
62. Zhang, B., Cheng, Z., Pedram, M.: A high-performance low-power barrett modular multiplier for cryptosystems. In: 2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED). pp. 1–6 (2021)
63. Zhu, H., He, J., Bowen, J.: From algebraic semantics to denotational semantics for Verilog. In: 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06). p. 341–360 (2006)

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

A Monads, Monad Transformers, and Reactive Resumption Monads over State in Haskell

This appendix includes background material on reactive resumption monads over state and, specifically, their representation in Haskell.

A.1 Monads in Haskell

A Haskell monad is a type constructor `m` with associated operations `return` and `>>=` with types:

```
return :: a → m a
(>>=)  :: m a → (a → m b) → m b
(>>)   :: m a → m b → m b   — "null" bind
x >> y  = x >>= λ_. y
```

A term of type `m a` is referred to as a *computation of* `a`, whereas a term of type `a` is a *value*—the distinction between values and computations is fundamental to monadic denotational semantics [37]. The `return` operation creates an `a`-computation from an `a`-value. The (`>>=`) operation is a kind of "backwards application" for `m`, meaning that, in `x >>= f`, an `a`-value is computed by `x` and then `f` is applied to that value. Null bind performs computation `x`, ignores its result, and then performs computation `y`.

Monadic return and bind operations are overloaded in Haskell and this overloading is resolved via the type class system.

The return and bind of a monad generally obey the "monad laws" that signify that `>>=` is associative and that `return` is a left and right unit of `>>=`. What makes monads useful in language semantics and functional programming, however, is not this basic infrastructure, but rather the other operations definable in terms of the monad (e.g., the state monad has operations for reading and writing to and from state).

A.2 `Identity` Monad

The type constructor for the identity monad is given by:

```
data Identity a = Identity a
```

It is conventional in Haskell to use `Identity` for both the type and data constructors for the identity monad. For `Identity`, return and bind are defined by:

```
return           = Identity
(Identity a) >>= f = f a
```

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

A.3 Monad Transformers in Haskell

A *monad transformer* is a construction `t` such that, for any monad `m`, `t m` is a monad. Monads created through applications of monad transformers to a base monad (e.g., `Identity`) are referred to as *modular* monads. For example, (`Re i s o`) from Fig. 4 is a modular monad; see Appendix A.6 below. For each monad transformer `t`, there is a lifting operation `lift :: m a → t m a` used to redefine `m`'s operations for `t m`.

A.4 `StateT` Monad Transformer

Return and bind for the monad `StateT s m` are defined in terms of `m`:

```
return a          = StateT (return_m a)
(StateT x) >>= f  = StateT (x >>=_m λ(a , s). deStateT (f a) s)
```

The return and bind operations are disambiguated by attaching an `m` subscript to `m`'s operations.

In addition to the standard return and bind operations, the state monad transformers also defines three other operations: `get` (to read the current state), `set` (to set the current state), and the overloaded `lift` (that redefines `m` computations as `StateT s m` computations):

```
get    :: StateT s m s
get    = StateT (λs. return_m (s , s))
set    :: s → (StateT s m ())
set s  = StateT (λ_. return_m (() , s))
lift   :: m a → StateT s m a
lift x = StateT (λs. x >>=_m λa. return_m (a , s))
```

A.5 `ReacT` Monad Transformer

The reactive resumption monad transformer is given by:

```
data ReacT i o m a = ReacT (m (Either a (o , i → ReacT i o m a)))
```

Return and bind for the monad `ReacT i o m` are defined in terms of `m`:

```
return a         = ReacT (return_m (Left a))
(ReacT x) >>= f  = ReacT (x >>=_m λr. case r of
                                        Left a       → f a
                                        Right (o , k) → return_m (o , λi. (k i) >>= f) )
```

The additional operations in `ReacT i o m` are:

```
signal   :: o → ReacT i o m i
signal o = ReacT (return_m (o , return)))
lift     :: m a → ReacT i o m a
lift x   = ReacT (x >>=_m (Left ∘ return_m))
```

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)

A.6 Reactive Resumption Monads over State in Haskell

In ReWire, device specifications have a type constructed using monad transformers defined above. The type constructor for devices is given by the type synonym `Re`—this is the Haskell definition equivalent to that from Fig. 4.

```
type Re i s o = ReacT i o (StateT s Identity)
```

ReWire allows a slightly more flexible formulation in which there are multiple `StateT` applications, although one such application as above suffices for the purposes of this work.

There are also projections from the monad transformer type constructors:

```
deStateT           :: StateT s m a → s → m (a, s)
deStateT (StateT x) = x
deReacT            :: ReacT i o m a → m (Either a (o , i → ReacT i o m a))
deReacT (ReacT x)   = x
```

Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited)