A Mechanized Semantic Metalanguage for High Level Synthesis

William L. Harrison
Two Six Technologies
Arlington, VA, USA
william.lawrence.harrison@gmail.com

Chris Hathhorn
Cyber Security Research Group
Oak Ridge National Laboratory
Oak Ridge, TN, USA
hathhorn@gmail.com

Gerard Allwein
US Naval Research Laboratory
Washington, DC, USA

ABSTRACT

High-level synthesis (HLS) seeks to make hardware development more like software development by adapting ideas from programming languages to hardware description and HLS from functional languages is usually motivated as a means of bringing software-like productivity to hardware development. Formalized semantics support a range of important capabilities in software languages (e.g., compositionality, comprehensibility, interoperability, formal methods, and security) that are desirable in hardware languages as well. This paper considers the formalized semantics of the Device Calculus, a typed λ-calculus with operators for constructing Mealy machines that forms a semantic substratum suitable for high-level synthesis and we demonstrate the utility of the Device Calculus as a foundation for formal methods in functional HLS with a case study specifying the semantics of an idealized subset of the FIRRTL language. FIRRTL ("Flexible Internal Representation for RTL") is an open-source hardware intermediate representation targeted by the Chisel hardware construction language and the semantics we present is also a starting point for exploring formal methods and security within both the Chisel toolchain and any other high-level synthesis flows that target FIRRTL.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; • **Security and privacy → Logic and verification**.

KEYWORDS

Formal aspects of hardware, Dependently-typed languages, High level synthesis, Formalized Semantics, Mechanized reasoning, Security foundations

ACM Reference Format:
William L. Harrison, Chris Hathhorn, and Gerard Allwein. 2021. A Mechanized Semantic Metalanguage for High Level Synthesis. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021), September 6–8, 2021, Tallinn, Estonia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3479394.3479417

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPDP 2021, September 6–8, 2021, Tallinn, Estonia
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8689-0/21/09…$15.00
https://doi.org/10.1145/3479394.3479417

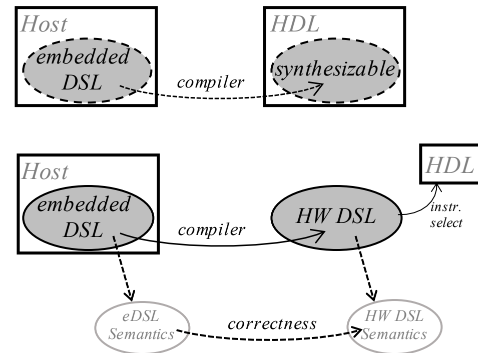


Figure 1: HLS compiler translates HLL Host subset to synthesizable subset of commodity HDL (top). Formulating compiler correctness requires bridging the software-hardware semantics gap (bottom).

1 INTRODUCTION

High-level synthesis (HLS) adapts software high-level languages (HLL) 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, 23, 56]. Herklotz and Wickerson [36] and Du et al. [19] 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 [48] and the state of the art is at a high-level of sophistication [43]. Are there necessary building blocks for verifying HLS compilers still required? This article answers the question in the affirmative, arguing for a semantic metalanguage that we call *Device Calculus* that is suitable to both source and target languages in an HLS flow.

Previous work [35] introduced Device Calculus as a means of capturing security design patterns for hardware; the current work is a companion piece laying out the formal underpinnings for that approach. The contributions of this paper are: (1) a formalized semantics for the Device Calculus; (2) a high-level overview of a rigorous semantics, formalized in Agda, for a substantial idealized subset of FIRRTL; and (3) feedback with respect to the FIRRTL language design that may prove useful for the Chisel/FIRRTL community. This paper is *not* intended to serve as a formal semantics for all of FIRRTL as it now exists—we leave that for future work—but the semantic case study we present demonstrates the viability of

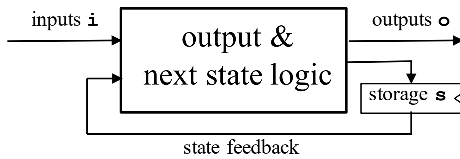


Figure 2: Mealy machines are a widely accepted intuition for guiding digital design [39, 46]. A formalization of Mealy machines as coinductive type (Fig. 3) underlies the semantics of the Device Calculus.

the approach and of the Device Calculus as building block in HLS verification.

Motivation. An HLS language may be viewed as a kind of embedded domain-specific language (eDSL) in which the embedding is literal (e.g., ReWire [53] is a subset of Haskell) or figurative (e.g., LegUp [13] is said to be "C-like"). Fig. 1 (top) shows the general structure of an HLS compiler, which translates the specific eDSL into a synthesizable subset of a commodity hardware description language (HDL) like Verilog or VHDL. Because formal semantics for commodity HDLs are problematic (see Section 1.1 below), this general structure would complicate the formal verification of an HLS compiler. Fig. 1 (bottom) shows the structure of an HLS compiler in which the code generation targets a machine-independent, hardware intermediate form (*HW DSL*). This restructuring is in one sense just good compiler engineering in that it makes the HLS compiler re-targetable—i.e., by delaying any commitment to machine- or HDL-dependent features until instruction selection. But, if *HW DSL* can be readily formalized, then it can play an important role in the formal verification of the entire HLS compiler, portrayed by the familiar compiler correctness diagram in Fig. 1 (bottom).

Figure 2 presents the graphical view of Mealy machines, that have been, and continue to be, the standard mental model in digital design practice [39, 46]—i.e., engineers "think" in terms of Mealy machines but program in commodity hardware definition languages when formulating a digital design. Upon any clock "tick" for the circuit, particular values of the input, register storage, and external outout (`i`, `s`, and `o`, resp.) are latched. During this clock period, the current values of `i` and `s` flow through the output and next-state logic—combinational circuitry—to produce new external output and storage values. This combinational logic is effectively a function of type $\mathtt{i} \to \mathtt{s} \to (\mathtt{o} \times \mathtt{s})$; that combinational logic is functional in nature has been recognized for many years [59] and forms the basis of most approaches to compiling functional programs to gates. These intuitions are formalized in Agda in Section 2.

This article argues the case for a mechanized semantic metalanguage suitable to HLS compiler verification. The Device Calculus is a typed λ-calculus with constants related to hardware description—e.g., types and operations for fixed size binary words as well as recursion operators for constructing Mealy machines. It appeared in previous work [35] as a metalanguage for security patterns for hardware, but this previous work did not explore the formalized semantics of Device Calculus. The current work presents these foundations mechanized in Agda along with a case study formalizing a significant portion of the open source, machine-independent, hardware intermediate representation called FIRRTL ("Flexible Internal Representation for RTL")[1].

What makes a metalanguage suitable for specifying HLS languages? "Harmony" with the underlying notion of computation of sequential hardware would seem to be a high-level answer. For example, hardware has a fixed memory footprint and this restricts, one way or another, the notions of program and data recursion in an HLS language [55]. Another difference between software programs and hardware designs generally is in termination behavior: hardware does not terminate. Hardware designers use Mealy machines of a particular, non-terminating form (see Fig. 2) as an organizing principle in digital design, and Mealy machines are at the heart of the Device Calculus metalanguage. The Device Calculus is a convenient λ-calculus syntax for constructing Mealy machines well-suited to the role of semantic metalanguage for HLS languages; λ-calculi have played that role in language semantics research for many decades [60]. We illustrate this suitability in Section 4 with the Idealized FIRRTL semantics. In most respects, the Device Calculus language definition is given in a familiar style of a denotational semantics encoding in dependently typed languages like Coq and Agda; this style of semantics is sometimes referred to as a *tagless* interpreter.

Section 1.1 describes related work. Section 2 describes the formalization of Mealy machines as a coinductive type in Agda. Section 3 presents the Device Calculus and its formalized semantics. Section 4 presents Idealized FIRRTL and its translation into the Device Calculus metalanguage. In order to make this article as self-contained as possible, this section includes an overview of FIRRTL as it exists [44]. Section 5 discusses conclusions and future directions.

1.1 Related Work

Mealy machines were first introduced to engineering practice as a kind of mental model to guide engineers in circuit design [46] and abstract state machines continue to inform the process [39]. Still, it is difficult to comprehend system level design in terms of low-level state machines and bridging this conceptual divide is a primary motivation behind high-level synthesis tools and languages. The most popular strategy adapts C-like languages: LegUp [13], Vivado HLS [68], FPM [66], Streams-C [26] and ROCCC [64]. Such tools confront the challenges of extracting coarse-grain parallelism from C-like languages [20]. High-level synthesis (HLS) based on languages and libraries for parallelism [14, 30] are common also.

There is a considerable literature on compiling functional programs to hardware [3, 4, 6, 21, 22, 24, 31, 51, 53, 59, 61] and this line of research was initially motivated by the desire to make hardware design more like software design: aspects of hardware's notion of computation (notably combinational circuitry) are inherently functional in nature. A parallel line of research in functional HLS [11, 12, 15–17, 23, 56, 57] seeks to adapt software formal methods for functional languages to verifying correctness and security properties of hardware designs and implementations. Both

[1] All the Agda code discussed here is available online [34]. Please note the notational convention at the beginning of Section 2.

```
record Mealy (i : Set) (s : Set) (o : Set) : Set where
  coinductive
  field fun : i → s → (o × s)
        now : i × s × o
        nxt : i → Mealy i s o

unfoldM : (i → s → (o × s)) → i → s → o → Mealy i s o
fun (unfoldM f i s o)   = f
now (unfoldM f i s o)   = (i , s , o)
nxt (unfoldM f _ s o) i = unfoldM f i s′ o′
            where (o′ , s′) = f i s

transcript : Mealy i s o → Stream i → Stream (i × s × o)
hd (transcript m is) = now m
tl (transcript m is) = transcript (nxt m (hd is)) (tl is)
runM : ℕ → Mealy i s o → Stream i → List (i × s × o)
runM n m is = takeStr n (transcript m is)
```

Figure 3: Mealy Machine Representation as Agda Coinductive Record, Corecursion Operator, & Stream Semantics

Nikhil [51] and Edwards [20] argue that conventional software languages are unsuitable for hardware design because of their adherence to a von Neumann programming model. Hardware parallelism is "massive, fine-grain, heterogeneous and reactive" and, as such, is fundamentally inharmonious with conventional languages like C, C++, etc. Nikhil then argues that Bluespec, a functional language based on term-rewriting, does fit well with hardware's notion of computation.

ReWire [29, 53, 56] is a functional hardware definition language embedded in the Haskell functional language. ReWire exhibits all of the benefits of a functional approach (e.g., expressiveness and concision) while allowing the specification of fine-grain concurrent algorithms and mechanisms. ReWire's language design is organized by a *reactive resumption monad* [53], a mathematical structure that models hardware parallelism which may be directly represented in Haskell. The Device Calculus was first identified as part of an ongoing effort to verify both ReWire applications and the ReWire compiler. Reynolds et al. [56] presented a small-step operational semantics for ReWire formalized in Coq. The research reported here grew out of that work and follow-on publications will report the formalization of ReWire in terms of Device Calculus. One of the primary motivations behind the current work is to build a foundation for a verified compilation process for ReWire.

Braibant and Chlipala [12] and Braibant [11] apply ideas from CompCert [43] to hardware synthesis and is the most closely related to ReWire. 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 [37]. 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 [53].

The organizing principle underlying the Device Calculus is a formulation of Mealy machines as a coinductive type in Agda. Coupet-Grimal and Jakubiec [16, 17] use a model for Mealy machines closely related to ours in their work in which Mealy machines are represented as stream-transducers. In their work, a Mealy machine is represented as a function of type `Stream i → s → Stream o`. This stream model of sequential circuitry is common enough—e.g., there are analogous constructions in Gordon [28], Zhu et al. [69], and Flor et al. [23]. The stream model views sequential circuitry—implicitly or explicitly—as a sequence of "snapshots" of a circuit's inputs, state, and outputs: the next state `s′` and output `o′` are determined by the current input `i` and a register state `s`. There is a "causality" hygiene condition that the semantics does not "look forward or backward" in the input stream (discussed in Flor et al. [23], page 12); i.e., that only the current state and a *single* input `i` determine the next state and output. The notion of causal stream functions is discussed by Uustalu and Vene [62]. The formulation of Mealy machines described below in Section 2 is factored through the `Mealy` coinductive type and the snapshot stream for a circuit is produced from `Mealy` by iteration across an input stream, thereby avoiding necessity of a causality side condition.

Gordon outlined the challenges of semantic specification of hardware definition languages [27], focusing specifically on Verilog, although his analysis applies equally to VHDL. There have been previous attempts to formalize VHDL as well [41] 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 [28], Lööw and Myreen [45], Khan [40], and Zhu, He and Bowen [69] do so for Verilog. Another approach is to design a new hardware language and compiler with formalization as a specific requirement. Examples of this approach are Kami [15] and CHERI [50]. Flor, Swierstra, and Sijsling presented Π-ware [23], a calculus of circuit combinators embedded in Agda with the goal of illustrating an approach to circuit design, testing, and formal verification within one language. Π-ware is organized around a single, designated circuit type with corresponding operators for constructing combinational and sequential circuit descriptions. Both Ghica and Jung [25] and Megacz [47] describe a categorical semantics for a class of digital circuits, the former using monoidal categories and the latter using generalized arrows. Device calculus, in contrast, is a conventional typed λ-calculus extended with a single type constructor for circuits (called `Dev`).

FIRRTL [44] is the target of the compiler for the Chisel hardware construction language and, because FIRRTL is machine-independent, it is key to making the Chisel compiler retargetable. The Chisel team argues forcefully that FIRRTL can be key to bringing software levels of productivity to hardware development [38]. The essence of their argument is that FIRRTL, if used as a common target for hardware language compiler front-ends, would enable library portability and language interoperability (i.e., something comparable, roughly speaking, to an LLVM (https://llvm.org) for hardware construction). One direction that has not been hitherto explored is the high assurance dimension for FIRRTL, and the Idealized FIRRTL semantics reported here provides a critical step in that direction.

Formal verification of (software) language compilers is a traditional area in programming languages research that has, of late, enjoyed considerable success with realistic compilers [43]. Compiler verification requires that both its source and target languages have rigorous mathematical semantics [67]. Compiler specification

```
_[]_       : Mealy i1 s1 o1 →
             Mealy i2 s2 o2 →
             Mealy (i1 × i2) (s1 × s2) (o1 × o2)
m1 [] m2 = unfoldM (fun× (fun m1) (fun m2))
                   (i1 , i2) (s1 , s2) (o1 , o2)
   where
       (i1 , s1 , o1) = now m1
       (i2 , s2 , o2) = now m2

feedbackM : (o1 → o2)
          → (i2 → o1 → i1)
          → i2
          → Mealy i1 s o1
          → Mealy i2 s o2
feedbackM out conn i2 m = unfoldM f2 i2 s (out o1)
    where
      wrap : (o1 → o2)
           → (i2 → o1 → i1)
           → o1
           → (i1 → s → (o1 × s))
           → (i2 → s → (o2 × s))
      wrap out conn oi f ix s = . . .
      (_ , s , o1)           = now m
      f2                     = wrap out conn o1 (fun m)

pipeline  : Mealy i1 s1 o1
          → Mealy o1 s2 o2
          → Mealy i1 (s1 × s2) o2
pipeline m1 m2 = feedbackM π2
                           (λ i1 → λ (o1 , o2) → (i1 , o1))
                           (π13 (now m1)) (m1 [] m2)
```

Figure 4: Mealy Operators for parallel, output feedback, and horizontal composition defined with `unfoldM`**. Function** `fun×` **is the function product and projection notation (i.e., π_2 and π^{13}) is explained in Section 2.**

is generally posed in the form of a commuting diagram (as in Fig. 1, bottom). Generally, a compiler correctness specification will require that, for a source program p, the meaning of p according to the source semantics will be related to the target semantics of the compiled code for p. Making the source and target semantics as "close" as possible—admittedly more of an imprecise, aesthetic judgment than a mathematical one—simplifies compiler verifications. We envision Device Calculus—a λ-calculus with a mostly "textbook" semantics—as a means of organizing HLS compiler verifications by bridging software and hardware semantics.

There has been surprisingly little formal methods research on multiple clock domains in functional HLS; to the authors' best knowledge, Czeck et al. [18] remains the only such published research. There has been substantial research in formal methods for multi-clock hardware in other language paradigms: Esterel [5], Lustre [32], and Signal [42]. Esterel is a synchronous imperative language, while Lustre and Signal are synchronous dataflow languages. Each of these languages has a rich formal methods literature, semantics, and toolsets, but it is their handling of clock zones or domains that is of particular relevance to functional HLS and reconfigurable computing generally.

```
W4 = BitVector 4
genfib4 : (Bit × Bit) → W4 × W4 → (W4 × (W4 × W4))
genfib4 (_ , 1) (_ , _) = (0000 , (0000 , 0001))
genfib4 (0 , _) (n , m) = (n , (n , m))
genfib4 (1 , _) (n , m) = (m , (m , n ⊞ m))
fib4 : Mealy (Bit × Bit) (W4 × W4) W4
fib4 = unfoldM genfib4 (0 , 0) (0000,0000) 0000
```

```
((0 , 0) , (0000 , 0000) , 0000) ::
((0 , 1) , (0000 , 0001) , 0000) ::
((1 , 0) , (0001 , 0001) , 0001) ::
((0 , 0) , (0001 , 0001) , 0001) ::
((1 , 0) , (0001 , 0010) , 0001) ::
((0 , 0) , (0001 , 0010) , 0001) ::
((1 , 0) , (0010 , 0011) , 0010) ::
((0 , 0) , (0010 , 0011) , 0010) ::
((1 , 0) , (0011 , 0101) , 0011) ::
((0 , 0) , (0011 , 0101) , 0011) ::
((1 , 0) , (0101 , 1000) , 0101) ::
((0 , 1) , (0000 , 0001) , 0000) :: []
```

Figure 5: Simple Example: Fibonacci as a Mealy Machine (top) with sample execution (bottom).

Designing hardware with multiple clock domains is a notoriously tricky design element to implement correctly with current reconfigurable computing toolchains. A multiple clock domain design is partitioned into several, non-overlapping clock zones or domains with each zone possessing its own notion of clock and clock domain intercommunication (i.e., routing signals between zones) requires some form of synchronization. In practice, Clock Domain Crossing (CDC) is accomplished with a range of synchronization mechanisms, including simple two flip-flop synchronizer to FIFO queues between domains [63]. Recent extensions to the C-based LegUp high-level synthesis tool [54] include synchronizing queues as source language constructs. Esterel, Lustre, and Signal provide a variety of language abstractions for clock types (e.g., discrete vs. continuous) as well as synchronization mechanisms. Recent work of Bourke et al. [8, 9] considers the formal verification of a Lustre compiler in Coq and, while this compiler targets software (i.e., assembly language), its formalization of timing is, we believe, also relevant to our work. Which forms of these abstractions make the most sense for Device Calculus and functional HLS generally is currently an open question of intense interest to the authors. We discuss this question further in Section 5.

2 COINDUCTIVE SEMANTICS FOR MEALY MACHINES

Notational Conventions. We will occasionally make syntactic simplifications to the Agda code presented in this paper. We will elide variable quantifications (e.g., "`∀ {i1 i2 s o1 o2} →`") to avoid visual clutter. Such simplifications will be made without further comment. Throughout we use a dependent projection notation, $\pi^{i\,n}$, which projects the ith component from an n-tuple. The codebase contains the full definitions [34].

The Mealy machine model (Fig. 2) can be neatly formalized as a coinductive record declaration in Agda (Fig. 3) that is the core of the Device Calculus semantics below in Section 3. The

```
data Type : Set where
  unit  : Type
  bit   : Type
  slv   : ℕ → Type -- standard logic vector a' la VHDL
  index : ℕ → Type
  _⇒_   : Type → Type → Type
  _⊗_   : Type → Type → Type
  _⊕_   : Type → Type → Type
  Dev   : Type → Type → Type → Type

Value : Type → Set
Value unit          = ⊤
Value bit           = Bool
Value (slv n)       = Vec Bit n
Value (index n)     = Fin n
Value (t1 ⇒ t2)     = (Value t1) → (Value t2)
Value (t1 ⊗ t2)     = (Value t1) × (Value t2)
Value (t1 ⊕ t2)     = (Value t1) ⊎ (Value t2)
Value (Dev i s o)   = Mealy (Value i) (Value s) (Value o)
```

Figure 6: Device Calculus: Type Syntax & Semantics in Summary.

record type `Mealy i s o` represents a Mealy machine with input, storage, and output types `i`, `s`, and `o` and its definition contains the type declaration for three methods, `fun`, `now`, and `nxt`. Each method type declaration includes `Mealy i s o` implicitly; e.g., the type of `now` is `Mealy i s o → i × s × o` rather than `i × s × o` as it appears in Fig. 3. The next state and output logic are defined by the function field, `fun : Mealy i s o → i → s → (o × s)`. The second field, `now`, defines the current values of the input, storage, and output. The third field defines the continuation function of the machine, `nxt : Mealy i s o → i → Mealy i s o`. For a machine `m` and input `i`, `(nxt m i)` is the continuation of `m` (in the sense of continuation-passing style in functional programming). The `unfoldM` operator constructs a `Mealy` machine given the first two fields (`fun` and `now`).

Fig. 3 presents a stream semantics for `Mealy` machines. The function `transcript` models a Mealy machine as a function from a stream of inputs to a stream of "snapshots" of type `(i × s × o)`. Each snapshot records the state of the input line, internal storage, and output line at each clock "tick". The definition of `transcript` uses Agda's copatterns [1] with the `Stream` type's destructors, `hd` and `tl` [34]. For convenience, we define a finite sampling function `runM` to generate finite snapshot lists for specific test cases.

Fig. 4 presents several operators for `Mealy` machines definable in terms of the `unfoldM` constructor. For machines `m1` and `m2`, `(m1 ⫴ m2)` creates a new machine in which `m1` and `m2` are placed in parallel. (⫴) is a synchronous parallelism operator in that each submachine is intuitively on the same "clock". `(feedbackM out conn i2 m)` can feedback the `o1` output as specified by a `conn` function. `(pipeline m1 m2)` connects `m1`'s output into `m2`'s input.

Figure 5 (top) presents a simple example of a `Mealy` machine, `fib4`. From the type of `fib4`, it takes a pair of `Bit`s, `(go, reset)`, as input on each "clock tick". If `reset` is set, it reinitializes registers `n` and `m` to `0000` and `0001`, respectively. Otherwise, if `go` is set, it updates `n` and `m` accordingly to calculate the Fibonacci sequence (here, ⊞ is unsigned addition on `W4`). `Mealy` machines can be "run" by mapping them into `Stream`s with an appropriate `Stream` of inputs; here, those inputs

```
data _⊩ᵈ_ : Cxt Type → Type → Set where
  var     : ∀ {Γ a} → a ∈ Γ → Γ ⊩ᵈ a
  lam     : ∀ {Γ a b} → (a :: Γ) ⊩ᵈ b → Γ ⊩ᵈ (a ⇒ b)
  app     : ∀ {Γ a b} → Γ ⊩ᵈ (a ⇒ b) → Γ ⊩ᵈ a → Γ ⊩ᵈ b
  _+_     : ∀ {Γ n}
          → Γ ⊩ᵈ (slv n) → Γ ⊩ᵈ (slv n)
          → Γ ⊩ᵈ (slv n)
  casebv  : ∀ {Γ n c}
          → Γ ⊩ᵈ (slv n) → Vec BitPat n
          → Γ ⊩ᵈ (slv n ⇒ c) → Γ ⊩ᵈ c → Γ ⊩ᵈ c
  (_,_)   : ∀ {Γ} {t1 t2 : Type}
            (a : Γ ⊩ᵈ t1) (b : Γ ⊩ᵈ t2)
          → Γ ⊩ᵈ (t1 ⊗ t2)
  π1      : ∀ {Γ} {t1 t2 : Type}
            (a : Γ ⊩ᵈ (t1 ⊗ t2))
          → (Γ ⊩ᵈ t1)
  π2      : ∀ {Γ} {t1 t2 : Type}
            (b : Γ ⊩ᵈ (t1 ⊗ t2))
          → (Γ ⊩ᵈ t2)
  inl     : ∀ {Γ} {t1 t2 : Type}
             (a : Γ ⊩ᵈ t1)
          → Γ ⊩ᵈ (t1 ⊕ t2)
  inr     : ∀ {Γ} {t1 t2 : Type}
            (b : Γ ⊩ᵈ t2)
          → Γ ⊩ᵈ (t1 ⊕ t2)
  𝟙       : ∀ {Γ} → Γ ⊩ᵈ unit
  0!      : ∀ {Γ} → Γ ⊩ᵈ bit
  1!      : ∀ {Γ} → Γ ⊩ᵈ bit
  ⟨_⟩     : ∀ {Γ n} → Vec Bit n → Γ ⊩ᵈ (slv n)
  ⦇_⦈     : ∀ {Γ n} → Fin n → Γ ⊩ᵈ (index n)
  -- Device level operations
  unfoldDev : ∀ {Γ i s o}
            → Γ ⊩ᵈ (i ⇒ s ⇒ (o ⊗ s))
            → Γ ⊩ᵈ i → Γ ⊩ᵈ s → Γ ⊩ᵈ o
              --------------------------
            → Γ ⊩ᵈ Dev i s o
  _||_     : ∀ {i1 s1 o1 i2 s2 o2 Γ}
           → Γ ⊩ᵈ Dev i1 s1 o1 → Γ ⊩ᵈ Dev i2 s2 o2
             ------------------------------------
           → Γ ⊩ᵈ Dev (i1 ⊗ i2) (s1 ⊗ s2) (o1 ⊗ o2)
  feedback : ∀ {i1 i2 s o1 o2 Γ}
           → Γ ⊩ᵈ (o1 ⇒ o2) → Γ ⊩ᵈ (i2 ⇒ o1 ⇒ i1)
           → Γ ⊩ᵈ i2 → Γ ⊩ᵈ Dev i1 s o1
             --------------------------
           → Γ ⊩ᵈ Dev i2 s o2
```

Figure 7: Device Calculus: Intrinsically-typed Syntax.

are of type `Stream (Bit × Bit)`. This `Stream` semantics for `Mealy` is given by the two functions, `transcript` and `runM`, defined in Fig. 3. Figure 5 (bottom) gives a sample run of `fib4`.

3 THE DEVICE CALCULUS AND ITS SEMANTICS

This section presents the Device Calculus and its formalization in Agda [34]. Fig. 6 presents the syntax and semantics of types in the Device Calculus. `Type` has constructors for functions (⇒), sums (⊕), and products (⊗). It also possesses hardware-relevant base types and type constructors—e.g., bit vectors or standard logic vectors (a.k.a., `slv`) in VHDL parlance—and related arithmetic and logical

```
⟦_⟧ : ∀ {Γ t} → (Γ ⊩d t) → Env Γ → Value t
⟦ var x ⟧ ρ           = lookup∈ ρ x
⟦ lam f ⟧ ρ           = λ s → ⟦ f ⟧ (s ▷ ρ)
⟦ app f e ⟧ ρ         = (⟦ f ⟧ ρ) (⟦ e ⟧ ρ)
⟦ e₁ + e₂ ⟧ ρ         = ⟦ e₁ ⟧ ρ ⊞ ⟦ e₂ ⟧ ρ
⟦ casebv e p f b ⟧ ρ = (⟦ f ⟧ ρ) $
                          (bitmatch p (⟦ e ⟧ ρ)) , (⟦ b ⟧ ρ)
   where
       _$_,_ : ∀ a b → (a → b) → Maybe a → b → b
       f $ nothing , b  = b
       f $ (just a) , _ = f a
⟦ ( a , b ) ⟧ ρ      = ( ⟦ a ⟧ ρ , ⟦ b ⟧ ρ )
⟦ π1 e ⟧ ρ            = Data.Product.proj₁ (⟦ e ⟧ ρ)
⟦ π2 e ⟧ ρ            = Data.Product.proj₂ (⟦ e ⟧ ρ)
⟦ inl e ⟧ ρ           = Data.Sum.inj₁ (⟦ e ⟧ ρ)
⟦ inr e ⟧ ρ           = Data.Sum.inj₂ (⟦ e ⟧ ρ)
⟦ 𝟙 ⟧ ρ               = tt
⟦ 0! ⟧ ρ              = Data.Bool.false
⟦ 1! ⟧ ρ              = Data.Bool.true
⟦ ⟨ v ⟩ ⟧ ρ           = v
⟦ ⦇ n ⦈ ⟧ ρ           = n
⟦ unfoldDev f i s o ⟧ ρ
    = unfoldM (⟦ f ⟧ ρ) (⟦ i ⟧ ρ) (⟦ s ⟧ ρ) (⟦ o ⟧ ρ)
⟦ d1 || d2 ⟧ ρ       = (⟦ d1 ⟧ ρ) ∥ (⟦ d2 ⟧ ρ)
⟦ feedback out conn dev i2 ⟧ ρ
    = feedbackM (⟦ out ⟧ ρ) (⟦ conn ⟧ ρ) (⟦ dev ⟧ ρ) (⟦ i2 ⟧ ρ)
```

Figure 8: Device Calculus: Semantics.

operations. The type constructor for devices, `Dev`, is intended to distinguish terms corresponding to `Mealy` machines and sequential circuitry. The `Value` function defines the semantics of `Type` in Fig. 6 making use of Agda's standard libraries for booleans, vectors, products, and sums (⊎). `Fin n` denotes a type of a set with `n` elements (defined in the standard library `Data.Fin`) and ⊤ is the unit type with single element `tt`.

The Device Calculus's intrinsically-typed syntax is given in Fig. 7. The style of the language's syntax encodes typing rules as data declarations in a manner familiar to users of Coq or Agda (e.g., see Wadler, Kokke, and Siek [65], Part 2). It is a de Bruijn style syntax, meaning that it is identifier-free; this obviates handling substitution, albeit at the expense of readability. De Bruijn style handles variables as references into a type environment (e.g., the "$a \in \Gamma$" in the `var` rule in Fig. 7). There are three constructors for `Dev` terms: `unfoldDev`, ||, and `feedback`. (+) signifies unsigned addition; there are a number of logical and arithmetic operations as well (see the codebase [34]).

The semantics of Device Calculus is given as a tagless interpreter in Fig. 8. The tagless style makes explicit use of Agda's dependent type system: given a term of type `t` and an appropriately typed environment, ⟦_⟧ returns a value of type `Value t`. (An excellent introduction to tagless interpreters may be found in Bove, Dybjer, and Norell [10]). Most of the semantic equations are unremarkable for a de Bruijn style λ-calculus. The device level operations correspond to the Mealy operators defined in Fig. 4; in particular, an `unfoldDev` term is mapped simply to the `unfoldM` constructor of `Mealy`. There are also bit patterns in the language with pattern-matching performed by `bitmatch`; e.g., a bit pattern ⓪⊛① matches any bit vector of length three starting and ending with bits `0` and `1`, resp. These definitions may be found in the codebase.

```
1  module MyModule :
2    input ina  : UInt<1>
3    input inb0 : UInt<2>
4    input inb1 : UInt<2>
5    input inb2 : UInt<2>
6    input clk  : Clock
7    output out : UInt<2>
8    wire c     : UInt<1>
9    c <= ina
10   reg r0     : UInt<2>, clk
11   reg r1     : UInt<2>, clk
12   reg r2     : UInt<2>, clk
13   r0 <= inb0
14   r1 <= mux(c, ina, inb1)
15   r2 <= inb2
16   out <= r0
```

Figure 9: Lowered FIRRTL Example ([44], p. 50).

4 IDEALIZED FIRRTL & ITS SEMANTICS

This section presents the formalized semantics of Idealized FIRRTL, specified as a syntactic translation into the Device Calculus semantic metalanguage. FIRRTL [44] is really a family of languages, and Section 4.1 describes the particular member of this family—"lowered" FIRRTL—that concerns us and, henceforth, we will refer to Idealized FIRRTL as IF when it is necessary to avoid confusion. Section 4.2 describes the syntax of IF as an intrinsically-typed syntax in Agda and motivates the syntax of IF's type language and, in particular, types for IF modules. Section 4.3 presents the semantic embedding of IF into Device Calculus. It is a common approach in language semantics to first translate the source language into an appropriate target λ-calculus and then to define that source language via the semantics of the target calculus; this is the approach we adopt.

4.1 Lowered FIRRTL Overview

Before proceeding further, we clarify first what we mean by "FIRRTL". We provide this background information on FIRRTL to motivate IF for readers unfamiliar with hardware definition languages, although it is not strictly necessary for understanding the remainder of the paper. FIRRTL [44] is a family of intermediate languages generated in succession by phases in the Chisel compiler. The Chisel compiler first generates high-level ("high") FIRRTL, and, through successive transformations, produces low-level ("lowered") FIRRTL. *Lowered FIRRTL* resembles a small subset of the Verilog HDL, and the Chisel compiler finally performs instruction selection from lowered FIRRTL into Verilog. It is lowered FIRRTL that first interested us because it is well-designed, machine-independent, and small (and, therefore, amenable to formalization).

To give the reader an idea of what lowered FIRRTL looks like, consider the example in Figure 9. This is a declaration of a single module that can be compiled into a standalone circuit (although it serves no purpose other than as a syntax example). FIRRTL is a typed language, with ground types for fixed-width unsigned and signed bit strings (`UInt<n>` and `SInt<n>`) and clocks (`Clock`).

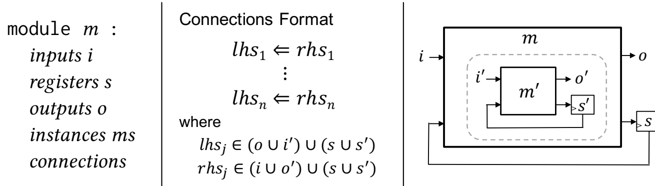


Figure 10: Idealized FIRRTL module syntax (left) with connections syntax and scoping (center). Module instances graphically (right) as a Mealy machine. These are described in detail in the text.

FIRRTL programs consist of declarations (resp., lines 2-8 and lines 10-12) and connect statements. For example, `ina` is an `input` of type `UInt<1>` (an unsigned bit string of width 1), and `r0` is a register (`reg`) of type `UInt<2>`, etc. Although it is not explicit in the syntax, each declaration has the form of a simple `let`; e.g., the scope of the `out` declaration on line 7 is the rest of the module (lines 8-16). A `wire` declaration is effectively a temporary `reg` declaration for semantic purposes and, hereafter, we identify the two forms.

The remainder of the module in Fig. 9 consists of connect statements of the form, `x <= e`. Each `output` and `reg` must be connected to—i.e., appear on the left-hand side of a connect—precisely once. The permissible right-hand sides are expressions involving `inputs` and `regs`. Modules may also contain instances of other modules (although the module in Fig. 9 does not) and the inputs and outputs of such instances also are in scope—more on this below. Connect statements appear at first glance to be a form of assignment, but they are really parts of function declaration, intuitively of type $inputs \rightarrow registers \rightarrow (outputs \times registers)$. The semantics described below in Section 4.3 makes this precise by translating `module` bodies into such functions explicitly.

Note that line 14 in the aforementioned code example contains what could be construed as a type error—i.e., `r1` (of width 2) being potentially connected to a value of width 1 (`ina`). Implicit conversions (e.g., converting `ina` to a width 2 bit-string) are eschewed by many languages—e.g., Haskell—because, although convenient, they render the code less clear and (potentially) could inadvertently obscure an error.

The one substantial restriction we make concerns the handling of clock signals that support multiple clock domains—specifically, our model assumes single clock circuit descriptions. Building FPGA applications with multiple clock domains (i.e., in which different parts of the circuit are driven by separate clocks) is well-known to be tricky in practice; we will return to the semantics of multiple clock domains in Section 5.

4.2 Syntax of Idealized FIRRTL

The Idealized FIRRTL (IF) language formalized in this section is an abstract form of lowered FIRRTL. We make certain mild assumptions about the syntax of FIRRTL circuit descriptions that do not, we believe, affect the generality or accuracy of the semantics we discuss below. For example, we only consider a representative subset of the expressions in FIRRTL as given in its definition [44].

The principal syntactic unit within IF is the module. Each IF module defines a synchronous circuit and, in the translation semantics

```
data GrTy : Set where
   UInt  : ℕ → GrTy
   SInt  : ℕ → GrTy
   Clock : GrTy

data Ty : Set where
  ↑_   : GrTy → Ty
  _⊗_ : Ty → Ty → Ty
  Unit : Ty

data ModTy : Set where
  Mod_,_,_⋊_ :
      Ty → Ty → Ty → Cxt ModTy → ModTy
```

Figure 11: Idealized FIRRTL: Types and Module Type Syntax

into the Device Calculus presented below, an IF module is mapped to a Device Calculus term of type `Dev i s o`. The challenge of specifying a typed syntax for IF lies mainly in determining scoping rules for the many varieties of binding that occur in the language. FIRRTL is a conventionally block-structured language in which, informally, a *circuit* has structure `c =` **let** $\mathtt{m}_1$ **in** $\cdots$ **let** $\mathtt{m}_n$ **in** `m` for modules $\mathtt{m}_j$ ($0 \leq j$) and `m` where the **let** is a simple (i.e., non-recursive) binder. FIRRTL leaves this let-structuring implicit, representing a circuit specification as a sequence of module definitions. IF makes this let-structuring explicit (Fig. 12).

The concrete syntax of a module has the form illustrated in Fig. 10 (left). The format of a FIRRTL module *m* consists of four declaration forms (i.e., of inputs *i*, registers *s*, outputs *o*, and instances *ms*) followed by sequences of instances and then sequences of connections (about which more will be said shortly). Each identifier declaration (e.g., "`input ina : UInt<1>`") is fairly self-explanatory. The FIRRTL specification [44] is less structured than the IF syntax–e.g., there is no order of declarations placing *inputs* before *outputs* and such declarations can occur within *connections* (e.g., Fig. 9 (left)). The *instances* declarations are a (possibly empty) sequence of previously defined modules; each module can be named multiple times as well. The final section includes *connections*. IF connections have the same syntax and scoping rules as FIRRTL connections.

Fig. 10 (center) gives scoping rules for IF/FIRRTL connections. Module instances—specifically, the inputs and outputs of instances—are also in scope in the connections of a module. Fig. 10 (right) illustrates this situation in which module *m* has an instance m'. Notice that the inputs of m'–i'–must also be connected to exactly once (i.e., appear on the left-hand side of "<=")—intuitively, this is necessary because m' is a kind of "sub-circuit" of *m* that must be fed inputs. For similar reasons, the outputs of m' are inputs available to *m*. The inputs (resp., outputs) of m' become outputs (resp., inputs) of *m*.

Fig. 11 presents the IF type syntax. The definition of simple types, `Ty`, is mostly self-explanatory. "Ground" types (using the terminology of the FIRRTL report [44]) are defined by `GrTy`, that has constructors for unsigned and signed integers and clock. The `Clock` type is included as a placeholder for later work and is not used in the current specification. `Ty` types are either `GrTy` or products.

```
data _⊢ᶜ_ :
   Cxt ModTy → ModTy → Set
 where
  mod    : ∀ {M Δ}
        →  M ⊢ᵐ Δ
           --------
        →  M ⊢ᶜ Δ
  letmod : ∀ {M Δ′}
        → (Δ : ModTy)
        → (M ⊢ᵐ Δ)
        → (Δ :: M) ⊢ᶜ Δ′
          ---------------
        → M ⊢ᶜ Δ′
```

Figure 12: Intrinsically-Typed Syntax of Idealized FIRRTL Circuits. Circuit declarations are sequences of module declarations with scoping akin to a simple (non-recursive) let. A type context (i.e., the type constructor Cxt) is a list in Agda.

A module type in Fig. 11 has the form, Mod i , s , o ⋊ ms. The Mod type constructor is similar to Dev from Figure 6, although it also contains a sequence of ModTy (represented as an Agda list) corresponding to the instances within a specific module. For example, the type of *m* from Fig. 10 would be Mod i , s , o ⋊ [Mod i′ , s′ , o′ ⋊ []] (assuming *m′* contains no instances).

Fig. 12 presents the syntax for IF circuits. As previously discussed, IF circuits are explicitly let-structured to clarify the circuit scoping. Circuit judgments have the form $M \vdash^c \Delta$, where M is a ModTy context and Δ is a ModTy. The letmod rule is simply a non-recursive let-binding specialized to the IF type system. Module judgments ($M \vdash^m \Delta$) are not defined in Fig. 12 but may be found in the codebase [34].

Fig. 13 presents a subset of the rules for IF expressions. The judgment form, $\Gamma \vdash_X a \,//\, \Delta$, includes a GrTy type environment, an GrTy, and a ModTy Δ. This figure displays the four variable reference forms in IF and a conditional expression (mux). Recall that de Bruijn style calculi are identifier-free and instead variable references are represented by a pointer into a type environment. For example, a reference to a register variable is given by a pointer into regs Δ (i.e., the s in Mod i , s , o ⋊ ms). The inp and node references are similarly defined. A qualified reference refers to an output of a module instance and the appropriate environment is calculated by the function $\mathtt{insts}_o$. A mux is a conditional expression where the condition contains a single bit.

4.3 Mealy Semantics for Idealized FIRRTL

Fig. 15 presents an overview of the translation semantics for Idealized FIRRTL into the Device Calculus. There are three main functions in the translation are trΔ, trC, and trM, the first two of which are defined in the figure; the type signature for trM is given. For Δ = Mod i , s , o ⋊ ms, inputs Δ, registers Δ, and outputs Δ are the translation into Type of i, s, and o, respectively.

The essence of the hardware formal model encapsulated by the Device Calculus views synchronous circuits as calls of the form, unfoldM f i s o. The semantics for Idealized FIRRTL extracts

```
data _⊢X_//_ : Cxt GrTy → GrTy → ModTy → Set
 where
  reg  : ∀ {Γ Δ a}
      → a ∈ regs Δ
        -----------
      →  Γ ⊢X a // Δ
  inp  : ∀ {Γ Δ a}
      → a ∈ ins Δ
        -----------
      → Γ ⊢X a // Δ
  node : ∀ {Γ Δ a}
      → a ∈ Γ
        ------------
      → Γ ⊢X a // Δ
  qexp : ∀ {Γ Δ a}
      → (μ : Ty)
      → μ ∈ instsₒ Δ
      → a ∈ μ
        ---------------
      → Γ ⊢X a // Δ
  mux  :  ∀ {Γ Δ a}
      →  Γ ⊢X UInt 1 // Δ
      →  Γ ⊢X a // Δ
      →  Γ ⊢X a // Δ
         -----------------
      →  Γ ⊢X a // Δ
```

Figure 13: Intrinsically-Typed Syntax of Idealized FIRRTL Expressions. The expression syntax pictured here reflects four kinds of variables in IF/FIRRTL. Full specification is available online [34].

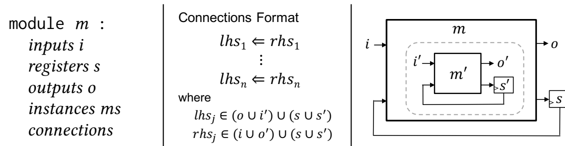
these arguments (i.e., f, i, s, and o) via a syntax-directed translation from an Idealized FIRRTL circuit declarations into the Device Calculus described in Section 3 previously. The translation trΔ calculates the Device Calculus types (i.e., Type) of these arguments. The function trM translates a module (i.e., its third argument $M \vdash^m \Delta$) into a closed Device Calculus term (i.e., $[\,] \Vdash^d$ trΔ Δ) with terms corresponding to a call to unfoldM. The function trC translates an Idealized FIRRTL circuit (i.e., its argument of type $M \vdash^c \Delta$) to produce a closed Device Calculus term (i.e., of type $[\,] \Vdash^d$ Dev (inputs Δ) (registers Δ) (outputs Δ)). The semantics of an Idealized FIRRTL circuit is then given by a composition of trC and the Device Calculus semantics, ⟦−⟧, defined in Fig. 8.

4.3.1 Type Language Translation. Fig. 14 shows the type translation from IF to Device Calculus. The translation of Ty is unremarkable, noting, however, that for the case of uni-clock circuit descriptions that we have assumed, the choice of Clock representation is irrelevant, and so, we map (↑ Clock) to bit. The ˝ty function calculates a product type in the IF Ty language. For an IF module of type Δ = Mod i , s , o ⋊ ms, the types of its registers will be a product of its own register type, s, with the register types of its modules ms; the helper function tystate computes this product. Modules and circuits of type Δ will be modeled in Device Calculus by a term of

type `Dev (inputs Δ) (registers Δ) (outputs Δ)` as described in Section 4.3.2, where the `inputs`, `registers`, and `outputs` functions are defined in Fig. 14.

4.3.2 Translating Modules. This section describes the translation of IF modules into Device Calculus, considering first the translation of modules without instance modules and then the translation of modules with instance modules.

Recall that the structure of Idealized FIRRTL modules was illustrated in Figure 10 and, for the reader's convenience, we repeat that figure here below. There is displayed on the left the general format of an Idealized FIRRTL module m and instance list ms. For the sake of this description, assume `m` has `ModTy` type `Mod i , s , o ⋊ ms`. This section describes translating `m` for the cases in which its module instances, `ms`, is empty and when it is non-empty.



Each module and module instance is specified ultimately as a `Dev`-typed Device Calculus term of the form of `unfoldDev f i∅ s∅ o∅`. Module `m` is specified as a Device calculus term of type `Dev i s o`. (Here, we conflate the IF `Ty` types `i`, `s`, and `o` with their translations into Device Calculus `Type` types for the sake of simplicity.) The expressions `i∅`, `s∅`, and `o∅` are the initial values of `m`'s inputs, registers, and outputs (assumed to be zero-ed bit vectors) and `f` is the transition function of type `i ⇒ s ⇒ (o ⊗ s)`. The function `trΔ` in Fig. 15 translates the type of m into the types of the Device Calculus terms `f`, `i∅`, `s∅`, and `o∅` that are arguments to `unfoldDev`.

Consider first the case in which m has no instances (i.e., ms is empty), then the scoping rules for connections (middle, above) are considerably simplified. The left-hand sides (lhs_j) are declared in either m's own outputs and registers (i.e., o and s) while the right-hand sides (rhs_j) are declared in either m's own inputs and registers (i.e., i and s). Therefore, in the case where ms is empty, the connections in module m determine in a straightforward manner a function of type $i \rightarrow s \rightarrow (o \times s)$ that is tantamount to the definition of `f`. In other words, the Device Calculus function `f` derives immediately from the syntax of m's connect statements in a straightforward syntax-directed fashion.

For the case in which ms is non-empty, it suffices to consider the case where there is precisely one module instance m' in ms. To see why this is, let ms be a sequence of m_i where $n \geq i > 1$. Each module instance m_i will be translated to a corresponding device, d_i, in the Device Calculus (i.e., a `Dev`-typed term). These multiple devices may be composed into a single device, $d_1 \,||\, \cdots \,||\, d_n$ with the Device Calculus parallelism operator ($||$). Therefore, assume there is precisely one module instance m' in ms.

The module instance m' has its own register state s' and the scoping rules for connections (pictured above) show that this state is directly visible within m—i.e., the registers in m' may be directly connected to within m. While this global "flat" view of state is straightforward enough to specify in Device Calculus, it does complicate the translation of modules defined by `trM`. The full definition of the module translation function `trM` is omitted from Fig. 15 but is available in the codebase [34].

```
-- Translate IF types to Device Calculus Types
trTy : Ty → Type
trTy (↑ UInt n) = slv n
trTy (↑ SInt n) = slv n
trTy (↑ Clock)  = bit
trTy (t1 ⊗ t2)  = (trTy t1) ⊗ trTy t2
trTy Unit       = unit

-- Helper function computing IF product type
Πty : Ty → Cxt Ty → Ty
Πty h []       = h
Πty h (t ∷ []) = h ⊗ t
Πty h (t ∷ ts) = h ⊗ (Πty t ts)

-- Calculate Register Ty from ModTy
mutual
  tystate : ModTy → Ty
  tystate (Mod _ , s , _ ⋊ []) = s
  tystate (Mod _ , s , _ ⋊ ms) = Πty s (tystates ms)
  tystates : Cxt ModTy → Cxt Ty
  tystates []      = []
  tystates (m ∷ μ) = tystate m ∷ (tystates μ)

-- Project/Translate from ModTy
inputs    : ModTy → Type
inputs Δ    = trTy (ins Δ)
   where ins : ModTy → Ty
         ins (Mod i , _ , _ ⋊ _)   = i
registers : ModTy → Type
registers Δ = trTy (tystate Δ)
outputs   : ModTy → Type
outputs Δ   = trTy (outs Δ)
   where outs : ModTy → Ty
         outs (Mod _ , _ , o ⋊ _)  = o
```

Figure 14: Idealized FIRRTL Type Semantics via Translation into the Device Calculus. See Section 4.3.1 for discussion.

4.3.3 The Running Example. Figure 16 (top) returns to the simple example presented first in Fig. 5. This example is a FIRRTL encoding of the Fibonacci circuit, `firfib4`; `firfib4` has type Δ according to the type system for Idealized FIRRTL. The example in Figure 16 (top) uses concrete syntax for readability's sake. Δ is a `ModTy`, which is a type constructor in the Idealized FIRRTL type that captures a module's input, internal storage, and output types (`(Bit ⊗ Bit)` , `(W4 ⊗ W4)` , and `W4`, respectively). The fourth field in a `ModTy` indicates the `ModTy` types of any instance modules within a module itself; for `firfib4`, there are no such instances.

The semantics of `firfib4`, `fib4′` in Figure 16 (right), is defined by composing the translation of FIRRTL circuits (given by `trC`) with the semantics of Device Calculus (⟦_⟧). The `Stream` semantics of `fib4′` produces the same output as that of `fib4` from Figure 5 as one would expect.

```
-- UnfoldDev argument Types
trΔ :  ModTy → Type
trΔ Δ = (i ⇒ s ⇒ (o ⊗ s)) ⊗ (i ⊗ s ⊗ o)
  where
     i = inputs Δ
     s = registers Δ
     o = outputs Δ

-- Module Translation
trM : ∀ M
    → (Δ : ModTy)
    → ModEnv M
    → M ⊢ᵐ Δ
    → [] ⊩ᵈ trΔ Δ
trM = ...

-- Circuit Translation
trC : ∀ {M}
    → (Δ : ModTy)
    → (ModEnv M)
    → (M ⊢ᶜ Δ)
    → ([] ⊩ᵈ Dev (inputs Δ) (registers Δ) (outputs Δ))
trC Δ ρ (mod x) = unfoldDev (π¹⁴ ans) (π²⁴ ans)
                            (π³⁴ ans) (π⁴⁴ ans)
  where
    ans = trM Δ ρ x
trC Δ ρ (letmod Δₓ x m) = trC Δ (trM Δₓ ρ x ▷ ρ) m
```

Figure 15: Idealized FIRRTL Circuit Semantics via Translation into the Device Calculus. See Section 4.3.2 for a detailed discussion of `trM`**.**

```
firfib4 : [] ⊢ᶜ Δ
module firfib4 :
  input go    : UInt<1>
  input reset : UInt<1>
  input clk   : Clock
  reg n       : UInt<4>, clk
  reg m       : UInt<4>, clk
  output out  : UInt<4>
  n <= if reset then 0
       elsif go then m else n
  m <= if reset then 1
       elsif go then n + m else m
  out <= n
----------------------------------------------
Δ : ModTy
Δ = Mod (Bit ⊗ Bit) , (W4 ⊗ W4) , W4 ⋊ []

fib4′ : Mealy
         (Vec Bit 1 × Vec Bit 1)
         (Vec Bit 4 × Vec Bit 4)
         (Vec Bit 4)
fib4′ = ⟦ trC Δ [] firfib4 ⟧ []
```

Figure 16: Fibonacci Revisited. This example creates the same stream as the previous example in Fig. 5.

4.4 Conformance of IF to FIRRTL

This section discusses the relationship between (lowered) FIRRTL as it exists now and Idealized FIRRTL as we have modeled it here. FIRRTL does not presently have a formal semantics and so there is not a standard formalization against which to compare the Idealized FIRRTL semantics described above. It is described at the FIRRTL project website (https://www.chisel-lang.org/firrtl/), its code repository (https://github.com/chipsalliance/firrtl), and in two publications [38, 44]. There is also an experimental simulator for lowered FIRRTL (henceforth, the "simulator") made available (https://github.com/freechipsproject/firrtl-interpreter) by the Chisel team.

One basis for comparison, therefore, is to encode example lowered FIRRTL programs in the IF syntax and compare the Agda output to that of the simulator. This experiment was successfully performed using sample lowered FIRRTL programs downloaded from its repository in the past. The experiment is described here and the full details of the testing (e.g., the lowered FIRRTL source code, etc.) are included in the codebase [34]. A complementary approach to conformance evaluation is outlined as well that involves proving the equivalence of the IF semantics to a suitable small-step operational semantics.

Experimental Evaluation. The aforementioned FIRRTL simulator includes an interactive shell to step through the execution of a lowered FIRRTL program. The first experiment that we performed was to encode the Fibonacci function running example described in Figures 5 and 16 in lowered FIRRTL and, then, to step through it with the simulator. The output for one step in this experiment, for example, is (pretty-printed slightly by hand):

```
CircuitState
Inputs          : clk = 0, go = 1, reset = 0
Outputs         : out = 1
Registers       : m = 2, n = 1
FutureRegisters: m = 3, n = 2
```

This simulator output corresponds to the transition between the sixth and seventh stream elements in Fig. 5

```
((0    , 0) , (n=0001  , m=0010)  , out=0001)
((go=1 , 0) , (n'=0010 , m'=0011) , 0010)
```

in which the relevant fields are labelled to correspond to the simulator output. In this Fibonacci experiment, the output of the IF semantics conformed to the simulator output. Three other examples previously downloaded from the FIRRTL project repository were evaluated in the same manner and found to conform with the simulator output. The details of this experimental evaluation, including the simulator output and the FIRRTL programs tested, are included in the codebase [34].

This experiment provides some basic confidence that there is agreement between the semantics for idealized subset of FIRRTL described here and the intentions of the Chisel/FIRRTL team. It is not clear that the simulator is considered as a kind of "gold standard" for lowered FIRRTL, but it does provide at least a basis for comparison with the present work. The process itself is quite time-consuming, including the hand-encoding of terms in Agda, the stepping through executions with the simulator, and the comparison of the two outputs. More rigorous automated testing (which we leave for future work), would require constructing a parser for lowered FIRRTL to support automated translation into IF syntax in

```
i⟨_⟩ : ModTy → Set
i⟨ Δ ⟩ = Value (inputs Δ)
s⟨_⟩ : ModTy → Set
s⟨ Δ ⟩ = Value (registers Δ)
o⟨_⟩ : ModTy → Set
o⟨ Δ ⟩ = Value (outputs Δ)

circuit : ∀ (Δ : ModTy)
        → (c : [] ⊢ᶜ Δ)
        → Mealy i⟨ Δ ⟩ s⟨ Δ ⟩ o⟨ Δ ⟩
circuit Δ c = ⟦ trC Δ [] c ⟧ []

step : ∀ (Δ : ModTy)
       → (c : [] ⊢ᶜ Δ)
       → (i⟨ Δ ⟩ × s⟨ Δ ⟩ × o⟨ Δ ⟩)
       → i⟨ Δ ⟩
       → (i⟨ Δ ⟩ × s⟨ Δ ⟩ × o⟨ Δ ⟩)
step Δ c (i,s,o) i' = { }0

equivalence : ∀ (Δ : ModTy)
            → (c : [] ⊢ᶜ Δ)
            → (i : i⟨ Δ ⟩)
            → step Δ c (now (circuit Δ c)) i
                     ≡ now (nxt (circuit Δ c) i)
equivalence Δ c = { }1
```

Figure 17: Equivalence Statement in the Device Calculus Agda Development between `circuit` semantics and Small-Step Operational Semantics `step`.

Agda as well as instrumentation of the FIRRTL simulator to align its output with that of the Agda semantics.

Specifying Equivalence. One advantage of small-step operational semantics for languages is that they are more readily understood by non-experts in language semantics. The mechanized semantics for IF presented in previous sections is based in semantic structures and techniques that are, we believe, not well-known outside of the programming languages and formal methods communities. One approach to evaluating the conformance of the IF semantics to FIRRTL, therefore, would be to first formulate a small-step semantics for lowered FIRRTL, establish its conformance in collaboration with the Chisel/FIRRTL team, and then prove an equivalence between between the two semantics. This approach, including the appropriate statement of formal equivalence, is outlined here and left for future work.

Any small-step operational semantics will include a transition relation or function between configurations; in the case of the proposed FIRRTL operational semantics, a reasonable configuration will include the circuit program itself and the current values of its inputs, register state, and outputs. A similar small-step semantics was described in previous research by Reynolds et al. [56] for the ReWire language. In that ReWire semantics, the transition relation is deterministic, and so, similarly, the small-step semantics we propose here will be represent in Agda as a function (rather than a relation) that we will call `step` in Fig. 17. In this high-level presentation, only this top-level function will be described.

Figure 17 first introduces several useful abbreviations in Agda; three functions (`i⟨_⟩`, `s⟨_⟩`, and `o⟨_⟩`) map a `ModTy` to the Agda model of its input, register state, and output types, respectively, by composition. The translation semantics of IF circuits is abbreviated by the `circuit` function in Fig. 17; note that this function produces a `Mealy` machine. A type declaration of a transition function for a small-semantics for FIRRTL—called `step`— can be given in Agda as portrayed in the figure. This incorporates constructions from the IF mechanization and fits into the pattern discussed above. The "`{ }0`" is Agda notation indicating that the body of the `step` definition has not yet been given.

Fig. 17 presents a statement of equivalence between the IF semantics, `circuit`, and the proposed operational semantics, `step`. The intuition behind this statement (called `equivalence` in the figure) is as follows. Let `m` be be the `Mealy` machine given by the IF semantics for `c`; that is, let `m` be (`circuit` Δ `c`). If one `step`s circuit `c` from `m`'s current state (i.e., `now` (`circuit` Δ `c`)) for a new input `i`, one arrives at the identical state via `m`'s `nxt` function. Again, the "`{ }1`" notation indicates that this proof of `equivalence` has yet to be given.

5 FUTURE WORK & CONCLUSIONS

This article introduced a mechanized semantic metalanguage suitable for HLS, the Device Calculus, and demonstrated its application in a substantial case study specification for an idealized subset of the FIRRTL hardware language. The current work is also a companion piece to previous work [35] that lays out the formal underpinnings for that approach to security patterns for hardware.

The Device Calculus semantics is denotational in style and mechanized in Agda, in contrast to our previous work on the mechanized semantics of the ReWire functional hardware description language [56], which was structured as a small-step operational semantics in Coq. We were motivated in part to explore this approach because of Agda's elegant formulation of coinductive types [1]. Sequential circuitry's notion of computation is inherently nonterminating—i.e., such circuits, in principle, never terminate—and so coinductive types play an essential role in our semantics of Device Calculus. The `Mealy` coinductive type introduced in Section 2 is at the heart of the HLS formal methodology we advocate.

A follow-on publication will present the Device Calculus mechanization of the ReWire functional hardware description language and its compiler. ReWire designs are organized by reactive resumption monadic semantics [33, 53] and the ReWire compiler's main translation phase is called *purification* [52]. Purification is a source-to-source translation that translates monadic effects (i.e., "impure" operations typed in a reactive resumption monad over state) into a pure core λ-calculus that resembles the Device Calculus. As such, the Device Calculus semantics for the ReWire language will closely resemble the main phase of its HLS compiler, thereby aiding, we expect, the formal verification of the ReWire compiler. Another follow-on publication will describe verification logics based on the `Mealy` semantics. Satisfaction in these logics has the form, $\mathtt{M}; \sigma \models \varphi$, in which `M` is a `Mealy` machine, σ is a stream of inputs, and φ is a sentence in the logic. We are currently experimenting with temporal and hybrid logics in this style mechanized in Agda. In this scenario, the machine `M` effectively induces a Kripke semantics for the logic.

The Chisel/FIRRTL team hypothesize [38] that FIRRTL can play a role in making hardware development more like software development; their primary focus is on making hardware development more agile and flexible by adapting techniques from software compilers. Although FIRRTL was designed as a target for the Chisel hardware construction language, there is no reason why other high level synthesis flows could not target it as well. Indeed, we initially became interested in FIRRTL as a target for the ReWire functional hardware description language. FIRRTL—or something like it—could play a role something along the same lines as LLVM (https://llvm.org) has played in software—that is, as a common retargetable backend with support for high assurance. Were this to come to pass, formulating the semantic foundations of FIRRTL *now* and letting those foundations guide its further development makes considerable sense.

In reconfigurable computing, supply chain vulnerabilities loom large in the setting of mission-critical systems [58]. Third-party IP modules provide useful, reusable functionality for FPGA applications, for example, but they come in vendor proprietary formats that render safety, correctness, or security analysis effectively intractable. Engineers, nonetheless, are understandably reluctant to give up third party IP modules despite their opacity to analysis.

Even software binaries can be reverse-engineered if need be, but there is no analogous "IDA Pro" for vendor-specific, proprietary hardware IP modules. That FIRRTL is open—and, consequently, not in a proprietary format—could make it a valuable part of high-assurance HLS toolchains, because it could be the target of multiple HLS languages and compilers (i.e., not only Chisel and ReWire). Having a rigorous semantics for a low-level, machine-independent, open source hardware language like FIRRTL, then, provides an important foundation for formal methods on reconfigurable hardware because of the availability of FIRRTL IP and its susceptibility to formal analyses. There is a growing library of IP in Chisel (and, hence, in FIRRTL) that could alleviate part of the "third party IP" challenge. We believe that the semantics of Idealized FIRRTL presented here provides a significant first step towards FIRRTL becoming a useful foundation for high assurance hardware development.

Languages are not generally designed and implemented with formalization in mind. In formulating a rigorous semantics for a language born "in the wild", one may encounter aspects of the language that may impede its formalization—although, admittedly, such "impediments" may be a matter of taste. However, taking this qualitative feedback into account can help improve future versions of the language. The authors envision this paper serving as a bridge between, roughly speaking, the hardware development, formal methods, and programming languages research communities.

The FIRRTL language has a `clock` type whose values are treated as first-class. In FIRRTL, clocks are associated directly with individual registers rather than with circuits or modules and this can complicate synthesis (or render it impossible) in certain situations. For example, a connection assignment, `c <= and(a, b)`, is problematic if `a` and `b` are on different clocks because, intuitively, when is the intended value of the `and(a, b)` available for `c`? An alternative to this fine-grained timing is the more coarse-grained notion of *multiple clock domains* in which designs are partitioned into non-overlapping clock zones. Recent extensions to the LegUp high-level synthesis tool [54] provide a means of representing multiple clock domains in the LegUp source code along with a compiler that insert FIFO queues to implement clock domain crossing automatically.

One remaining semantic challenge for Device Calculus is accommodating multiple clock domains within a single device (i.e., term of type `Dev i s o`). The work on multiple clock domains in Esterel introduced by Berry and Sentovich [5] is of particular relevance here. In particular, they extend Esterel with a clock "sub-language" for expressing different clock domains; a similar language will be introduced into the Device Calculus through its type system through its extension as a type effect system [49]. Because clock domains are static, it makes sense to encapsulate them as effect labels in the Device Calculus type system. In such a scenario, one could imagine clock tags annotating each device and being tracked through parallel compositions; e.g., device types would now have the form $\mathtt{Dev}^{c}\ \mathtt{i}\ \mathtt{s}\ \mathtt{o}$ for clock tag c. Then, an asynchronous parallelism operator could be introduced; given $m_j : \mathtt{Dev}^{c_j}\ \mathtt{i}_j\ \mathtt{s}_j\ \mathtt{o}_j$, the typing rule would appear something like:

$$m_1 \mid\mid m_2 : \mathtt{Dev}^{(c_1 \sqcup c_2)}\ (\mathtt{i}_1 \otimes \mathtt{i}_2)\ (\mathtt{s}_1 \otimes \mathtt{s}_2)\ (\mathtt{o}_1 \otimes \mathtt{o}_2)$$

Here, the $(c_1 \sqcup c_2)$ clock tag indicates multiple (i.e., at least two) clock domains.

To complete this semantics for such an extension to Device Calculus would necessitate the development of a mechanized theory of asynchronous Mealy machines. In practice, communication between clock domains is typically handled by circuitry implementing an asynchronous queue. Indeed, recent extensions to the LegUp high-level synthesis tool [54] automatically generate linking circuitry (i.e., asynchronous queues) between different clock domains. Completing this scenario to extend the current Device Calculus semantics, then, would add a formal model of asynchronous queues to the current (uni-clock) Mealy semantics. `Clock` signals are really fundamentally different and (in our opinion) ought to be treated as such in the language.

In the parlance of language design, FIRRTL treats `Clock`s as first-class values. That is, in FIRRTL [44], the `Clock` type is treated syntactically as any other type—e.g., they can be passed into and out of modules as any other signal values. However, `Clock`s can be manipulated only by conversions to and from other bit vector types, there are no FIRRTL operations specific to that type other than conversions to and from `UInt` and `SInt` types; these are, specifically, the `asClock`, `asUInt`, and `asSInt` instructions (p.42-3, Li et al. [44]).

From a language design point of view, this situation would seem to be potentially problematic. For one thing, `Clock`s are static—circuits do not vary their clock rates. Unlike other types, for example, `Clock` timing must ultimately be determined statically by the compiler and synthesis tools. While synthesis tools can use static analyses to reject ill-defined programs, it would be better (certainly from a semantic point of view) to screen out such cases when possible via the language's type system itself. Considering the reputation of multiple clock domain engineering as tricky, language support for developing such applications is highly desirable. From a language design point of view, `Clock` signals are not first-class values in the ordinary sense of that term despite being passed as ordinary inputs and outputs.

REFERENCES

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th POPL*. 27–38.

[2] David Andrews. 2015. Will the Future Success of Reconfigurable Computing Require a Paradigm Shift in Our Research Community's Thinking? Keynote address, Applied Reconfigurable Computing.

[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] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *DAC*. 1216–1225.

[5] Gerard Berry and Ellen Sentovich. 2001. Multiclock Esterel. In *Correct Hardware Design and Verification Methods, CHARME 2001*, Vol. Lecture Notes in Computer Science, vol 2144. Springer.

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

[7] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. 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* (London, UK) *(PLDI 2020)*. 243–257.

[8] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. *SIGPLAN Not.* 52, 6 (June 2017), 586–601. https://doi.org/10.1145/3140587.3062358

[9] Timothy Bourke, Lélio Brun, and Marc Pouzet. 2019. Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset. *Proc. ACM Program. Lang.* 4, POPL, Article 44 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371112

[10] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types, Vol. 5674. 73–78.

[11] Thomas Braibant. 2011. Coquet: A Coq Library for Verifying Hardware. In *Certified Programs and Proofs*. 330–345.

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

[13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 24 (Sept. 2013), 27 pages.

[14] Jongsok Choi, S. Brown, and J. Anderson. 2013. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Field-Programmable Technology (FPT), 2013 International Conference on*. 270–277. https://doi.org/10.1109/FPT.2013.6718365

[15] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *PACMPL* 1 (2017), 24:1–24:30.

[16] Solange Coupet-Grimal and Line Jakubiec. 1999. Hardware Verification Using Co-induction in Coq. In *Proceedings 12th TPHOLs*. 91–108.

[17] Solange Coupet-Grimal and Line Jakubiec. 2004. Certifying Circuits in Type Theory. *Form. Asp. Comput.* 16, 4 (Nov. 2004), 352–373.

[18] E. Czeck, R. Nanavati, and J. Stoy. 2006. Reliable Design with Multiple Clock Domains. In *Proc. MEMOCODE*. 139–148.

[19] Zewei Du, Yann Herklotz, Nadesh Ramanathan, and John Wickerson. 2021. Fuzzing High-Level Synthesis Tools. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 148. https://doi.org/10.1145/3431920.3439466

[20] Stephen A. Edwards. 2006. The Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design & Test* 23 (2006), 2006.

[21] Stephen A. Edwards. 2013. *Functioning Hardware from Functional Programs*. Technical Report CUCS-027-13. Department of Computer Science, Columbia University.

[22] Stephen A. Edwards, Martha A. Kim, Richard Townsend, Kuangya Zhai, and Lianne Lairmore. 2019. *The FHW Project: High-Level Hardware Synthesis from Haskell Programs.* Technical Report CUCS-003-19. Department of Computer Science, Columbia University.

[23] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2015. Π-Ware: Hardware Description and Verification in Agda. In *Proc. TYPES*.

[24] Marco Gerards, Christiaan Baaij, Jan Kuper, and Matthijs Kooijman. 2011. Higher-Order Abstraction in Hardware Descriptions with CλaSH. In *Proceedings of the 2011 14th EUROMICRO Conference on Digital System Design (DSD '11)*. IEEE Computer Society, Washington, DC, USA, 495–502. https://doi.org/10.1109/DSD.2011.69

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

[26] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. 2000. Stream-oriented FPGA computing in the Streams-C high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*. 49–56.

[27] Michael J. C. Gordon. 1995. The semantic challenge of Verilog HDL. *Proc. of 10th Annual IEEE LICS* (1995), 136–145.

[28] Michael J. C. Gordon. 2002. Relating Event and Trace Semantics of Hardware Description Languages. *Comput. J.* 45, 1 (01 2002), 27–36.

[29] Ian Graves, Adam M. Procter, William Harrison, and Gerard Allwein. 2015. Provably Correct Development of Reconfigurable Hardware Designs via Equational Reasoning. In *FPT*. 160–171.

[30] David Greaves and Satnam Singh. 2008. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE Computer Society.

[31] Jim Grundy, Tom Melham, and John O'leary. 2006. A reflective functional language for hardware design and theorem proving. *J. Funct. Program.* 16, 2 (March 2006), 157–196.

[32] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

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

[34] William Harrison. 2021. Device Calculus Codebase. Available from https://www.dropbox.com/s/j121q0p87z9k90b/codebase.tar.gz?dl=1.

[35] William L. Harrison and Gerard Allwein. 2020. Verifiable Security Templates for Hardware. In *Proceedings of the Design, Automation, and Test Europe (DATE) Conference*.

[36] Yann Herklotz and John Wickerson. 2021. High-level synthesis tools should be proven correct. In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*.

[37] Bluespec Homepage. 2017. http://bluespec.com.

[38] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216.

[39] Randy H. Katz. 2000. *Contemporary Logic Design* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[40] Wilayat Khan, Alwen Tiu, and David Sanan. [n.d.]. VeriFormal: An Executable Formal Model of a Hardware Description Language.

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

[42] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. 1991. Programming real-time applications with SIGNAL. *Proc. IEEE* 79, 9 (1991), 1321–1336.

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

[44] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. 2016. *Specification for the FIRRTL Language.* Technical Report UCB/EECS-2016-9. EECS Department, University of California, Berkeley.

[45] Andreas Lööw and Magnus O. Myreen. 2019. A Proof-Producing Translator for Verilog Development in HOL. In *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 99–108.

[46] G. H. Mealy. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34, 5 (Sep. 1955), 1045–1079.

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

[48] F. Lockwood Morris. 1973. Advice on Structuring Compilers and Proving Them Correct. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 144–152. https://doi.org/10.1145/512927.512941

[49] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer.

[50] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1003–1020.

[51] Rishiyur S. Nikhil. 2011. Abstraction in Hardware System Design. *Commun. ACM* 54, 10 (Oct. 2011), 36–44. https://doi.org/10.1145/2001269.2001284

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

[53] Adam Procter, William Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. 2017. A Principled Approach to Secure Multi-core Processor Design with ReWire. *ACM TECS* 16, 2, Article 33 (Jan. 2017), 33:1–33:25 pages.

[54] O. Ragheb and J. H. Anderson. 2018. High-Level Synthesis of FPGA Circuits with Multiple Clock Domains. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 109–116.

[55] Thomas Reynolds, Rohit Chadha, William L. Harrison, and Gerard Allwein. 2020. Strongly Bounded Termination with Applications to Security and Hardware Synthesis. In *ACM Workshop on Type Driven Development (TyDe)*.

[56] Thomas N. Reynolds, Adam Procter, William Harrison, and Gerard Allwein. 2019. The Mechanized Marriage of Effects and Monads with Applications to High-assurance Hardware. *ACM TECS* 18, 1, Article 6 (Jan. 2019), 26 pages.

[57] Cherif Salama, Gregory Malecha, Walid Taha, Jim Grundy, and John O'Leary. 2011. Static consistency checking for Verilog wire interconnects—Using dependent types to check the sanity of Verilog descriptions. *Higher-Order and Symbolic Computation* 24, 1-2 (2011), 81–114.

[58] Raymond Shanahan. 2014. US Department of Defense Trusted Microelectronics. In the Proceedings of the 17th Annual NDIA Systems Engineering Conference.

[59] Mary Sheeran. 1984. muFP, A Language for VLSI Design. In *LISP and Functional Programming*. 104–112.

[60] Joseph E. Stoy. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press.

[61] George Ungureanu and Ingo Sander. 2017. A Layered Formal Framework for Modeling of Cyber-Physical Systems. In *Design Automation and Test in Europe (DATE 2017)*. Lausanne, Switzerland.

[62] Tarmo Uustalu and Varmo Vene. 2005. The essence of dataflow programming. In *Proceedings of the Third Asian conference on Programming Languages and Systems (APLAS'05)*. 2–18.

[63] Saurabh Verma and Ashima Dabare. 2007. Understanding Clock Domain Crossing Issues. *The EE Times* (24 Dec. 2007). https://www.eetimes.com/understanding-clock-domain-crossing-issues/

[64] J. Villarreal, A. Park, W. Najjar, and R. Halstead. 2010. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. 127–134. https://doi.org/10.1109/FCCM.2010.28

[65] Phillip Wadler, Wen Kokke, and Jeremy Siek. 2021. Programming Language Foundations in Agda. https://plfa.github.io.

[66] C. Wang, X. Li, J. Zhang, P. Chen, X. Feng, and X. Zhou. 2012. FPM: A Flexible Programming Model for MPSoC on FPGA. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*. 477–484.

[67] Glynn Winskel. 1993. *The formal semantics of programming languages: an introduction*. MIT Press.

[68] Xilinx Corporation 2021. Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[69] Huibiao Zhu, Jifeng He, and J. Bowen. 2006. From algebraic semantics to denotational semantics for Verilog. In *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)*. 341–360.