Strongly Bounded Termination with Applications to Security and Hardware Synthesis

Thomas Reynolds
University of Missouri
Columbia, Missouri
tom@k3s.com

William L. Harrison
Cyber Security Research Group
Oak Ridge, Tennessee
william.lawrence.harrison@gmail.com

Rohit Chadha
University of Missouri
Columbia, Missouri
chadhar@missouri.edu

Gerard Allwein
US Naval Research Laboratory
Washington, DC
gerard.allwein@nrl.navy.mil

Abstract

Termination checking is a classic static analysis, and, within this focus, there are type-based approaches that formalize termination analysis as type systems (i.e., so that all well-typed programs terminate). But there are situations where a stronger termination property (which we call strongly-bounded termination) must be determined and, accordingly, we explore this property via a variant of the simply-typed λ-calculus called the bounded-time λ-calculus (BTC). This paper presents the BTC and its semantics and metatheory through a Coq formalization. Important examples (e.g., hardware synthesis from functional languages and detection of covert timing channels) motivating strongly-bounded termination and BTC are described as well.

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

Keywords Language semantics, Termination analysis, Mechanized reasoning, Security foundations, High level synthesis

ACM Reference Format:
Thomas Reynolds, William L. Harrison, Rohit Chadha, and Gerard Allwein. 2020. Strongly Bounded Termination with Applications to Security and Hardware Synthesis. In *TyDe '20: ACM Workshop on Type Driven Development, Sunday, August 23, 2020, Jersey City, NJ*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

TyDe '20, August 23, 2020, Jersey City, NJ

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 Introduction

In the classic example of a covert timing channel, the value of b in `if b then` c_1 `else` c_2 leaks when c_i have different termination behavior. Determining that the program is timing-channel free, however, requires knowing more than whether both c_i terminate, but also that, when they both terminate, they do so in precisely the same number of steps. This is the basic insight underlying compiler strategies for eliminating timing channels [3]. The latter timing-aware notion of termination—which we call *strongly-bounded termination* (SBT)—is stronger than the one generally considered in the literature (e.g., Dershowitz and Manna [11] and their descendants).

We introduce the bounded-time λ-calculus (BTC), a variety of the simply-typed λ-calculus, in which the type system enforces SBT: all BTC terms of type t are guaranteed to terminate within a fixed number of steps encoded within the type t itself. This restricted expressiveness facilitates the Coq verification of interesting metatheoretic properties of our system—e.g., type safety and strong normalization [31]. In developing BTC, we are motivated primarily by two examples that require precise bounds on termination. The first example is the covert timing channel analysis [3] illustrated above. The second example concerns the high-level synthesis (HLS) of hardware circuits from functional languages that takes functions (e.g., in host languages like Haskell [4, 13, 17, 29] or Scala [5]) and attempts to compile them to hardware circuitry. The remainder of this section comments on related work. Section 2 presents an overview of BTC and its syntax and semantics. Section 3 presents its metatheory, all of which is formalized in Coq (source available upon request). Section 4 discusses our two motivating examples in light of the formal development of BTC and Section 5 outlines future work and conclusions.

Related Work. The ReWire functional hardware description language is a tool for producing high assurance hardware [29]. ReWire is a subset of the Haskell functional programming language: every ReWire program is a Haskell

program, but not necessarily vice versa. A recent publication presents a type-effect system for ReWire and a related operational semantics mechanized in Coq [32] and it is this formal ReWire semantics that is the original point of departure for BTC. The ReWire type-effect system accounts for space usage in a manner reminiscent of effect systems for region-based analysis [27], albeit in a restricted form. In ReWire, "regions" are of fixed size and number, and access to a "region" is controlled via the state monad transformer. Space usage in ReWire is also strictly bounded via restrictions on data and control recursion. BTC was developed as a means of exploring both the easing of recursion restrictions and the formalization of synthesizability for ReWire and the authors expect that BTC, in some form, will be integrated with the ReWire effect system soon.

Ghica and Jung [15] provide a categorical semantics for a class of digital circuits in terms of monoidal categories and are motivated by the need for supporting syntactic, equational reasoning. Another categorical presentation of digital circuits is found in Megacz [25], who uses generalized arrows as a basis for hardware description. Linear Types are used to measure and control resource usage in Brunel, et al. [8] and Ghica, et al. [16]. By contrast with this categorical approach, ReWire specifications are, more or less, ordinary functional programs that are compiled into circuits. ReWire specifications may be reasoned about equationally in the usual manner of functional languages; this was the approach taken in our previous ReWire verification work [20, 29].

Whereas other type-based termination analyses examine more expressive languages [9, 30], we focus on a limited extension of the simply-typed λ-calculus because of its relevance to HLS generally and ReWire in particular. The tradeoff in expressiveness facilitates the use of standard machinery to prove interesting metatheoretic properties of our system. To the best knowledge of the authors, the BTC formalization presented here is the first such type-based termination analysis to have been fully mechanized and verified in Coq.

Functional language HLS must determine if a function may be represented as circuitry. This question is non-trivial because compiling functional languages to hardware is not possible for arbitrary programs because hardware's finite storage capacity can accommodate neither unbounded data nor control. Functions that are to be represented as combinational circuitry must exhibit SBT (as we discuss in Section 4), which neither Haskell nor Scala's type systems can discern. BTC types have been augmented with natural numbers as a representation of computation time so that, for any BTC type t, there is an $n \in \mathbb{N}$ such that, for any closed term $e : t$, then e normalizes in no more than n steps. The natural n is, then, the notion of "size" of the type t in the BTC type system. Sized types were introduced as a means of performing type-based termination analysis [2, 6, 21, 26, 34] and this paper explores the sized type approach to strongly-bounded termination analysis and its formalization in Coq.

Type based approaches to termination add size parameters to type system as a means to guarantee that recursive functions terminate. The typing rule ListFix illustrates a (simplified) type-based approach to using size variables in recursive definitions (adapted from [6, 34]):

$$\frac{\Gamma, f : [a]^{\boldsymbol{n}} \to b \vdash e : [a]^{\boldsymbol{n}+1} \to b}{\Gamma \vdash \mathtt{fix}(\lambda f.e) : [a]^{\infty} \to b} \text{ (ListFix)}$$

where $\boldsymbol{n}$ is a size variable and $[a]^{\boldsymbol{n}}$ denotes the type of lists (with elements of type a) of a size no greater than $\boldsymbol{n}$. This requires each instance of f to be defined on lists smaller than e, and hence, each recursive call reduces the size parameter.

Using types as a basis for termination analysis dates as far back as Mendler [26]; see Abel [1] and Sacchini [33] for full discussion of this idea. The underlying motivation for using sized types is that it aids in termination checking, as subsequent calls may be type checked for reduced size. Hughes et al. [21] incorporate sized types into a functional language. In the system introduced in [21] each name for a datatype, i.e., `List`, `Stream`, represents a collection of nat-indexed datatypes such as $\mathtt{List}^n$ where n is a size bound. In this system, sizes are a linear function of size variables and typing rules reinforce a requirement that each input generates an output of a smaller size. This supports a basic check for responsiveness of programs in a reactive system because programs that are well-typed in this system will satisfy a *liveness* property–that every input eventually produces an output.

Building on the system introduced in [21], Pareto [28] examines an extension of Haskell with sized types. This extension utilizes linear sized types–including addition and constants and provides a type-checking algorithm as well. Giménez [18] considers an extension of the Calculus of Constructions [10] in which sizes are not explicitly represented but are still present nonetheless. Other type systems involve more complex size algebras. For example, a more expressive language using linear sized types was introduced in [34] by extending the Calculus of Inductive Constructions with (co-)inductive types and size annotations. Other systems introduce sizes as upper bounds [2, 7], or add sized types in a dependently typed framework with polymorphism and indexed types [39]. Each of these systems has more expressive power than our own.

Functional language approaches to hardware description and synthesis frequently take the form of domain-specific languages embedded in a general purpose functional language like Haskell or Scala [4, 5, 13, 17, 29]. With this approach, one must distinguish between programs that describe hardware (i.e., those in the embedded DSL) and those that do not (i.e., host functional programs that cannot be represented in hardware), but that distinction is not made formal in previous work. This work identifies SBT as an important property prior to hardware realizability and formalizes its basic type-theoretic machinery.

Type-theoretic machinery has been used in timing channel analysis in the context of imperative programs (see, for example, [14, 36, 37, 40]). These approaches typically combine information flow analysis with timing channel analysis. For information flow, the program variables are attached to security levels, and the type systems ensure that information does not flow from high-level security variables into low-level security variables. For reasoning about timing channels analysis, Smith and Volpano [36, 37] assume that each reduction of operational semantics takes a single time unit. By assuming that time is an explicit low-level security variable which is assigned to inside a program, Smith and Volpano [37] can exploit the information flow analysis to show that well-typed programs are free of timing channels. On the other hand, Smith [36] explicitly decorates each instruction type with the number of steps it takes to execute the instruction. Thus, the approach of Smith [36] is analogous to ours, except that it applies to imperative programs. The type system in Zhang et al. [40] identifies fragments of code that may have timing channels when implemented in hardware. The identification allows the programmers to mitigate timing channels by making sure that these fragments take a fixed amount of time in hardware. In Ferraiuolo et al. [14], a secure hardware-description language, ChiselFlow, is described with type annotations that inform a custom-made processor architecture, HyperFlow, when to mitigate information leakage (including timing leakage).

2 The Bounded Time Calculus

In this section, we introduce a variant of the simply-typed λ-calculus—called the bounded-time λ-calculus (BTC)—with a sized type system enforcing strongly-bounded termination. The BTC type system has been augmented with natural numbers representing a coarse grained approximation of computation time. This calculus extends a standard Church-style type system of the simply-typed λ-calculus in two ways. First, in the BTC type system, function types are the only types themselves that have timing decorations in a manner similar to the ListFix example above. Other types, such as products, sums and unit remain unchanged. Second, type judgments incorporate a timing decoration as part of the judgment. The formalization contains proofs of many standard properties of the simply-typed λ-calculus such as type safety and strong normalization. The Coq development is available from the codebase [31]; every theorem, lemma, and corollary in this paper has been verified in Coq.

Syntax. The BTC type, term, and value syntax is just that of the simply-typed λ-calculus with one exception. The function type is decorated with a natural number; e.g., $(T \xrightarrow{n} U)$. The decoration on the function type represents a restriction on the time it takes to convert an argument to its corresponding output and we will return to its significance below. The type syntax is stated below in Definition 2.1. We adopt the following conventions. We use s, t, u to denote terms, v, w to denote values, $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ to denote arbitrary variables, and T, U for types.

Definition 2.1 (Types)**.** The set Ty of BTC types is defined thusly:

$$T, U \in Ty ::= T \xrightarrow{\boldsymbol{n}} U \mid T \times U \mid T + U \mid ()$$

where $\boldsymbol{n}$ denotes an arbitrary natural number.

Terms and values are given standard definitions (Definition 2.2).

Definition 2.2 (Terms)**.** The set *term* of BTC terms is defined thusly:

$$\begin{aligned} s, t, u \in \mathit{term} ::= {} & \boldsymbol{x} \mid \mathit{app}\ t\ u \mid \lambda\, \boldsymbol{x}\ T\ t \mid \mathtt{nil} \mid \mathit{pair}\ t\ u \\ & \mid \pi_1\ t \mid \pi_2\ t \mid \mathit{inl}\ t\ T \mid \mathit{inr}\ t\ T \mid \mathit{case}\ s\ t\ u \end{aligned}$$

Definition 2.3 (Values)**.** The set *value* of BTC values is given by the following:

$$v, w \in \mathit{value} ::= \lambda\, \boldsymbol{x}\ T\ t \mid \mathtt{nil} \mid \mathit{pair}\ v\ w \mid \mathit{inl}\ v\ T \mid \mathit{inr}\ v\ T$$

The definitions of free variable and substitution (Definitions 2.4 and 2.5) are standard and given below. Term t is *closed* when $\boldsymbol{FV}(t) = \emptyset$.

Definition 2.4 (Free Variables)**.** For any term t, the set of free variables in t, $FV(t)$ is defined as:

$$\begin{aligned} \boldsymbol{FV}(x) &= \{x\} \\ \boldsymbol{FV}(\mathit{app}\ t\ u) &= \boldsymbol{FV}(t) \cup \boldsymbol{FV}(u) \\ \boldsymbol{FV}(\lambda\ x\ T\ t) &= \boldsymbol{FV}(t) \setminus \{x\} \\ \boldsymbol{FV}(\mathit{pair}\ t\ u) &= \boldsymbol{FV}(t) \cup \boldsymbol{FV}(u) \\ \boldsymbol{FV}(\mathit{case}\ s\ t\ u) &= \boldsymbol{FV}(s) \cup \boldsymbol{FV}(t) \cup \boldsymbol{FV}(u) \\ \boldsymbol{FV}(\mathtt{nil}) &= \{\} \\ \boldsymbol{FV}(\pi_1\ t) &= \boldsymbol{FV}(t) \\ \boldsymbol{FV}(\pi_2\ t) &= \boldsymbol{FV}(t) \\ \boldsymbol{FV}(\mathit{inr}\ t\ U) &= \boldsymbol{FV}(t) \\ \boldsymbol{FV}(\mathit{inl}\ u\ T) &= \boldsymbol{FV}(u) \end{aligned}$$

Definition 2.5 (Substitution)**.** Substitution of value v for free occurrences of x in t is defined by:

$$\begin{aligned} y[x := v] &= \begin{cases} y & y \neq x \\ v & \text{otherwise} \end{cases} \\ (\mathit{app}\ t\ u)[x := v] &= \mathit{app}\,(t[x := v])\,(u[x := v]) \\ \mathtt{nil}[x := v] &= \mathtt{nil} \\ (\lambda\, x\ T\ t)[x := v] &= \begin{cases} \lambda\, y\ T\,(t[x := v]) & y \neq x,\ y \notin \boldsymbol{FV}(v) \\ \lambda\, x\ T\ t & \text{otherwise} \end{cases} \\ (\mathit{pair}\ t\ u)[x := v] &= \mathit{pair}(t[x := v])\,(u[x := v]) \\ (\pi_1\ t)[x := v] &= \pi_1\,(t[x := v]) \\ (\pi_2\ t)[x := v] &= \pi_2\,(t[x := v]) \\ (\mathit{inr}\ t\ U)[x := v] &= \mathit{inr}\,(t[x := v])\,U \\ (\mathit{inl}\ u\ T)[x := v] &= \mathit{inl}\,(u[x := v])\,T \\ (\mathit{case}\ s\ t\ u)[x := v] &= \\ & \mathit{case}\,(s[x := v])\,(t[x := v])\,(u[x := v]) \end{aligned}$$

Type System. Typing rules for terms are given in Definition 2.6. Typing judgments take the form $\Gamma \vdash t : T^n$ where $\Gamma = \{\boldsymbol{x}_1 : T_1, \ldots, \boldsymbol{x}_m : T_m\}$ such that for each assumption $\boldsymbol{x}_i : T_i$, $\boldsymbol{x}_i$ denotes a term variable unique to Γ and T_i is a type (as defined in Definition 2.1). The set Γ is commonly referred to as a *context* or *environment*. In cases where Γ is empty, we write $\{\}$. Additionally, we write $\Gamma, \boldsymbol{x} : T$ as shorthand for $\Gamma \cup \{\boldsymbol{x} : T\}$. The syntax of decorators is given by the following grammar:

$$n, m \in \mathbb{N} ::= n \mid n + m \mid \max(n, m)$$

We say that n decorates the type T in T^n. Though restrictive, this linear structure suffices for our needs here. The expression that decorates function types is more restrictive—only allowing natural numbers as decorators.

Definition 2.6 (Type Inference System)**.**

$$\frac{}{\Gamma, \boldsymbol{x} : T \vdash \boldsymbol{x} : T^0}\ (\textsc{Var}) \qquad \frac{\Gamma, \boldsymbol{x} : T \vdash t : U^n}{\Gamma \vdash \lambda\, \boldsymbol{x}\ T\ t : (T \xrightarrow{n} U)^0}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash f : (T \xrightarrow{n} U)^m \ \Gamma \vdash t : T^p}{\Gamma \vdash \mathit{app}\ f\ t : U^{(n+m+p+1)}}\ (\textsc{App}) \qquad \frac{}{\Gamma \vdash \mathtt{nil} : ()^0}\ (\textsc{Nil})$$

$$\frac{\Gamma \vdash t : T^n \ \Gamma \vdash u : U^m}{\Gamma \vdash \mathit{pair}\ t\ u : (T \times U)^{n+m}}\ (\textsc{Pair})$$

$$\frac{\Gamma \vdash t : (T \times U)^n}{\Gamma \vdash \pi_1\ t : T^{(n+1)}}\ (\textsc{Pi}_1) \qquad \frac{\Gamma \vdash t : (T \times U)^n}{\Gamma \vdash \pi_2\ t : U^{(n+1)}}\ (\textsc{Pi}_2)$$

$$\frac{\Gamma \vdash t : T^n}{\Gamma \vdash \mathit{inl}\ t\ U : (T + U)^n}\ (\textsc{Inl}) \qquad \frac{\Gamma \vdash u : U^n}{\Gamma \vdash \mathit{inr}\ u\ T : (T + U)^n}\ (\textsc{Inr})$$

$$\frac{\Gamma \vdash s : (T + U)^n \ \Gamma \vdash t : (T \xrightarrow{l} S)^m \ \Gamma \vdash t : (U \xrightarrow{r} S)^p}{\Gamma \vdash \mathit{case}\ s\ t\ u : S^{(n+\max(l+m, r+p)+2)}}\ (\textsc{Case})$$

The types for variables, abstractions and `nil` each have 0 as a decorator. Pairs inherit the sum of the decorators for the types of their subterms, while each type for the projection constructors adds one to the decorator of their subterm types. Sums possess the same decorators as the types of their subterms. For the application rule, the natural number decorating the arrow represents the time it takes to reduce a term of type T to a term of type U. In addition to natural number indices, function types also receive an outer decoration. This represents the time for processing the function of that type. In line with the other term constructors, the decorator for the resulting term adds 1. The rule for *case* takes the max value of the decorators adorning either branch of the evaluation. Because this requires an additional term, it adds 2 to the decorator for the type of the case expression.

The type system possesses a property common to many simply-typed λ-calculi. This property is that every well-typed term has a unique type, as stated in Theorem 2.7.

Theorem 2.7 (Type Uniqueness). *If $\Gamma \vdash t : T^n$ and $\Gamma \vdash t : U^m$, then $T = U$.*

In this typing system, with the addition of decorators, this property was not guaranteed. Interestingly, the type system also possesses similar property for decorators, as stated in Theorem 2.8. This property enforces a uniformity of decorator assignments, so to speak.

Theorem 2.8 (Decorator Uniqueness). *If $\Gamma \vdash t : T^n$ and $\Gamma \vdash t : U^m$, then $n = m$.*

Additionally, well-typed terms in the empty context are well-typed in any context. Theorem 2.9 provides further assurance that the addition of decorators does not drastically alter the traditional properties of the simply-typed λ-calculus's type system.

Theorem 2.9. *If $\{\} \vdash t : T^n$, then $\Gamma \vdash t : T^n$.*

Our type system and definition of values (from Definition 2.3) provide us with *canonical forms*—that is, a property of closed, well-typed values. Many proofs of metatheoretic properties tend to be organized around canonical forms. This greatly reduces the cases one needs to consider. Canonical forms are given by Lemmas 2.10-2.13 below.

Lemma 2.10. *If $\{\} \vdash v : (T \xrightarrow{n} U)^m$ and v is a value, then there exists x and u such that $v = \lambda\, x\ T\ u$.*

Lemma 2.11. *If $\{\} \vdash v : (T \times U)^m$ and v is a value, then there exists t and u such that $v = \mathit{pair}\ t\ u$.*

Lemma 2.12. *If $\{\} \vdash v : (T + U)^m$ and v is a value, then there exists w such that $v = \mathit{inl}\ w\ U$ or $v = \mathit{inr}\ w\ T$.*

Lemma 2.13. *If $\{\} \vdash v : ()^0$, then $v = \mathtt{nil}$.*

Substitution (given in Definition 2.5 above) preserves typing judgments. This requires that if free variables occur in well-typed terms, then there must be a typing assignment for those variables relative to the context (Lemma 2.14 below).

Lemma 2.14. *If $x \in \mathbf{FV}(t)$ and $\Gamma \vdash t : T^n$, then there exists a U such that $\{x : U\} \in \Gamma$.*

From this Corollary 2.15 follows—namely, that a term is closed if it is well-typed in the empty context.

Corollary 2.15. *If $\{\} \vdash t : T^n$, then t is closed.*

Moreover, we have Lemma 2.16 as a consequence—that the context of a typing judgment does not alter typing judgments, so long as all each context maintains assignments of types to any free variable.

Lemma 2.16. *If $\Gamma \vdash t : T^n$ and, if, for all x, $x \in \mathbf{FV}(t)$, Γ and Γ' assign the same type to x, then $\Gamma' \vdash t : T^n$.*

Finally, we have Theorem 2.17—that is, the substitution operation preserves typing judgments when the term being substituted is a value.

Theorem 2.17. *If $\Gamma, x : U \vdash t : T^n$, value v, and $\{\} \vdash v : U^m$, then $\Gamma \vdash (t[x := v]) : T^n$.*

This is a more restricted version than what one typically sees. In most simply-typed λ-calculi, no additional restriction is placed on terms being substituted into expressions. Our version adds the restriction that a value must be substituted. In theory, all that one needs is to restrict the decorator of the type for such terms as in Corollary 2.18.

Corollary 2.18. *If* $\Gamma, x : U \vdash t : T^n$ *and* $\{\} \vdash v : U^0$, *then* $\Gamma \vdash (t[x := v]) : T^n$.

In practice, no proof hinges on which version one picks and the reason for this is simple. In BTC, all well-typed values have 0 as their decorator (Theorem 2.19).

Theorem 2.19. *If* $\{\} \vdash t : T^n$ *and value* v, *then* $n = 0$.

Small-Step Operational Semantics. In this section, we describe a small-step operational semantics for BTC mechanized in Coq in the style of the popular Software Foundations series. The single-step reduction relation ($\rightsquigarrow$) is given by the rules in Definition 2.20 below.

Definition 2.20 (Step Relation)**.**

$$\frac{value\ v}{app\ (\lambda\, \boldsymbol{x}\ T\ t)\ v \rightsquigarrow [\boldsymbol{x} := v]t}\ (\text{ST\_AppAbs})$$

$$\frac{t \rightsquigarrow t'}{app\ t\ u \rightsquigarrow app\ t'\ u}\ (\text{ST\_App1}) \qquad \frac{value\ v \quad u \rightsquigarrow u'}{app\ v\ u \rightsquigarrow app\ v\ u'}\ (\text{ST\_App2})$$

$$\frac{t \rightsquigarrow t'}{pair\ t\ u \rightsquigarrow pair\ t'\ u}\ (\text{ST\_Pair1}) \qquad \frac{value\ v \quad u \rightsquigarrow u'}{pair\ v\ u \rightsquigarrow pair\ v\ u'}\ (\text{ST\_Pair2})$$

$$\frac{value\ v \quad value\ w}{\pi_1\ (pair\ v\ w) \rightsquigarrow v}\ (\text{ST\_Pi1}) \qquad \frac{value\ v \quad value\ w}{\pi_2\ (pair\ v\ w) \rightsquigarrow w}\ (\text{ST\_Pi2})$$

$$\frac{t \rightsquigarrow t'}{\pi_1\ t \rightsquigarrow \pi_1\ t'}\ (\text{ST\_Pi1E}) \qquad \frac{t \rightsquigarrow t'}{\pi_2\ t \rightsquigarrow \pi_2\ t'}\ (\text{ST\_Pi2E})$$

$$\frac{t \rightsquigarrow t'}{inl\ t\ T \rightsquigarrow inl\ t'\ T}\ (\text{ST\_Inl}) \qquad \frac{t \rightsquigarrow t'}{inr\ t\ T \rightsquigarrow inr\ t'\ T}\ (\text{ST\_Inr})$$

$$\frac{s \rightsquigarrow s'}{case\ s\ t\ u \rightsquigarrow case\ s'\ t\ u}\ (\text{ST\_Case})$$

$$\frac{value\ v}{case\ (inl\ v\ T)\ t\ u \rightsquigarrow app\ t\ v}\ (\text{ST\_CaseL})$$

$$\frac{value\ v}{case\ (inr\ v\ T)\ t\ u \rightsquigarrow app\ u\ v}\ (\text{ST\_CaseR})$$

The step relation has a useful property that is immediately provable:

Theorem 2.21 (Deterministic Evaluation)**.** *If* $s \rightsquigarrow t$ *and* $s \rightsquigarrow u$, *then* $t = u$.

As stated in Theorem 2.21, this property is that the BTC step relation is deterministic. We discuss more properties of our semantics in the next section.

3 Metatheory

In this section we discuss the metatheoretic properties of BTC. In particular, *type safety* (Section 3.1) and *strong normalization* (Section 3.2) are covered. In standard approaches to proving metatheoretic properties of simply-typed λ-calculi, it is common to define an additional step relation. This is typically the reflexive-transitive closure of the single step relation. Because the reflexive-transitive closure provides no information on the number of steps taken, we do not take this approach. Our interests demand something different. In our setting, we use $\overset{\boldsymbol{n}}{\rightsquigarrow}$ to denote a natural number indexed extension of our step relation, stated in Definition 3.1.

Definition 3.1 (Nat Indexed Step Relation)**.**

$$\frac{}{t \overset{0}{\rightsquigarrow} t}\ (\text{Refl}) \qquad \frac{s \rightsquigarrow t \quad t \overset{\boldsymbol{n}}{\rightsquigarrow} u}{s \overset{\boldsymbol{n+1}}{\rightsquigarrow} u}\ (\text{Step})$$

This relation has many useful properties. The most important of which are stated in Lemma 3.2 and Theorem 3.3.

Lemma 3.2. *For* s, t, u *and* i, j, *we have the following properties of the indexed step relation:*

(Inclusion) *If* $t \rightsquigarrow u$, *then* $t \overset{1}{\rightsquigarrow} u$,

(Transitivity) *If* $s \overset{i}{\rightsquigarrow} t$ *and* $t \overset{j}{\rightsquigarrow} u$, *then* $s \overset{i+j}{\rightsquigarrow} u$.

The first property is an inclusion property—it tells us that the indexed relation includes $\rightsquigarrow$. The second property is a transitivity property—it tells us that indexed relation is transitive and that the indices are additive. Each of these properties and Definition 3.1 is used to prove Theorem 3.3.

Theorem 3.3 (Congruence)**.** *For each rule stated in Definition 2.20, there exists a corresponding version with* $\rightsquigarrow$ *replaced by* $\overset{\boldsymbol{n}}{\rightsquigarrow}$. *For rules* ST_AppAbs, ST_Pi$_1$, ST_Pi$_2$, ST_CaseL, *and* ST_CaseR, $\rightsquigarrow$ *is replaced by* $\overset{1}{\rightsquigarrow}$.

3.1 Type Safety

In small-step operational semantics, type safety is the combination of two properties: *progress* and *preservation*. Traditionally speaking, the former is the property that all well-typed terms are either values or they step to some other term. In our setting, we incorporate decorators into the mix. In the case of progress (Theorem 3.4), decorators play no additional role.

Theorem 3.4 (Progress)**.** *If* $\{\} \vdash t : T^n$, *then either* t *is a value or there exists* u *such that* $t \rightsquigarrow u$.

The same cannot be said of preservation (Theorem 3.5).

Theorem 3.5 (Preservation)**.** *If* $\{\} \vdash t : T^n$ *and* $t \rightsquigarrow u$, *then there exists* m *such that* $m < n$ *and* $\{\} \vdash u : T^m$.

When well-typed terms take a step, the decorator for the type of the term stepped-to must be strictly smaller than that of the decorator for the term stepped-from. That is, preservation guarantees a reduction in decorators.

When we replace the $\rightsquigarrow$ with its indexed counterpart, we gain a variant of preservation (stated in Corollary 3.6) that relates decorators to the natural number indexes for the indexed step relation.

Corollary 3.6. *If* $\{\} \vdash t : T^n$ *and* $t \overset{m}{\rightsquigarrow} u$, *then there exists* l *such that* $l + m \leq n$ *and* $\{\} \vdash u : T^l$.

This tells us that as a term reduces, the resulting decorator for its type has an upper bound determined by its initial decorator minus the number of steps taken.

Progress and preservation guarantee that well-typed terms never "get stuck," so to speak. That is to say, for any term t, if t cannot step (by some application of rules from Definition 2.20), and t is not a value, then something has gone wrong in the process of computing t. Theorems 3.4 and 3.5 guarantee that this situation will not arise with well-typed terms.

Corollary 3.7 (Soundness). *If $\{\} \vdash t : T^n$ and $t \rightsquigarrow u$, then u is either a value or there exists v such that $u \rightsquigarrow v$.*

3.2 Strong Normalization

Normalization is a property of the step relation—often stated in terms of possible sequences of steps in the reduction of terms. A step (or reduction) relation is *weakly normalizing* if there exists a finite sequence steps ending in a *normal-form*—an irreducible term. If every such sequence ends in a normal-form, we say that the step relation is *strongly normalizing*. In BTC, all values are normal-forms, so we use "value" in place of "normal-form" without any issues. Because evaluation in BTC is deterministic, proving strong normalization is equivalent to proving that BTC is terminating.

Though not every λ-calculus is strongly normalizing, BTC is and we show this by establishing, using methods discovered independently by Tait [38] and Girard [19], that all well-typed BTC terms *terminate* (in the sense stated in Definition 3.8).

Definition 3.8 (Termination). For any term t, t *terminates* iff there exists v, n such that $t \overset{n}{\rightsquigarrow} v$ and v is a value.

Our step and indexed step relations preserve termination (as stated in Lemma 3.9).

Lemma 3.9. *For all terms t, u,*

1. *If $t \rightsquigarrow u$, then t terminates iff u terminates.*
2. *If $t \overset{n}{\rightsquigarrow} u$, then t terminates iff u terminates.*

Definition 3.10 defines our notion of reducibility sets. The final clause for unit types is included only for completeness. Because only `nil` has unit as its type, and `nil` is a value, `nil` terminates since we have $\texttt{nil} \overset{0}{\rightsquigarrow} \texttt{nil}$.

Definition 3.10 (Reducibility Sets). For any term t, such that $\{\} \vdash t : T^n$ and t terminates, $t \in \boldsymbol{R}_T$ is determined by T:

$$\frac{(T \text{ is } U \overset{m}{\to} V) \quad t \in \boldsymbol{R}_{(U \overset{m}{\to} V)}}{\forall w, \text{ if } w \in \boldsymbol{R}_U, \text{then } (app\, t\ w) \in \boldsymbol{R}_V}$$

$$\frac{(T \text{ is } U \times V) \quad t \in \boldsymbol{R}_{(U \times V)}}{\exists m\, w, \mathit{value}\ w, t \overset{m}{\rightsquigarrow} w, \pi_1(w) \in \boldsymbol{R}_U \ \&\ \pi_2(w) \in \boldsymbol{R}_V}$$

$$\frac{(T \text{ is } ()) \quad t \in \boldsymbol{R}_{()}}{\exists m\, w, \mathit{value}\ w \ \&\ t \overset{m}{\rightsquigarrow} w}$$

$$\frac{(T \text{ is } U + V) \quad t \in \boldsymbol{R}_{(U+V)}}{\begin{array}{l}\exists m\, w, \mathit{value}\ w, \quad (t \overset{m}{\rightsquigarrow} \mathit{inl}\ w\ U \ \&\ w \in \boldsymbol{R}_V) \\ \qquad\qquad \vee\ (t \overset{m}{\rightsquigarrow} \mathit{inr}\ w\ V \ \&\ w \in \boldsymbol{R}_U)\end{array}}$$

For BTC if we had base, or atomic types, we would add the following clause to Definition 3.10; this clause for atomic types and the clause for unit types are equivalent: T is *atomic* means $t \in \boldsymbol{R}_T$ *iff* t terminates. Lemma 3.11 collects some facts about reducibility sets—$\boldsymbol{R}$ sets for short—that follow from Definition 3.10. Girard [19] refers to the properties enumerated in Lemma 3.11 instead as conditions on reducibility sets—named 'CR' properties. Ours differ slightly, but remain close in spirit.

Lemma 3.11. *For all terms t, u arbitrary n, m, and T,*

1. *If $t \in \boldsymbol{R}_T$, then t terminates,*
2. *If $t \in \boldsymbol{R}_T$, then there exists l such that $\{\} \vdash t : T^l$,*
3. *If $t \rightsquigarrow u$ and $t \in \boldsymbol{R}_T$, then $u \in \boldsymbol{R}_T$,*
4. *If $t \overset{n}{\rightsquigarrow} u$ and $t \in \boldsymbol{R}_T$, then $u \in \boldsymbol{R}_T$,*
5. *If $\{\} \vdash t : T^m$, $t \rightsquigarrow u$ and $u \in \boldsymbol{R}_T$, then $t \in \boldsymbol{R}_T$,*
6. *If $\{\} \vdash t : T^m$, $t \overset{n}{\rightsquigarrow} u$ and $u \in \boldsymbol{R}_T$, then $t \in \boldsymbol{R}_T$.*

From Lemma 3.12—the $\boldsymbol{R}$-Substitution Lemma— it follows that the BTC is strongly normalizing. This lemma is more commonly referred to as the "Substitution Lemma."

Lemma 3.12 ($\boldsymbol{R}$-Substitution). *Let $v_1, \ldots, v_n$ be values such that for each $i = \{1, \ldots, n\}$, $v_i \in \boldsymbol{R}_{V_i}$. If $\{x_1 : V_1, \ldots, x_n : V_n\} \vdash t : T^j$, then $(t[x_1 := v_1] \ldots [x_n := v_n]) \in \boldsymbol{R}_T$.*

By property 2 of Lemma 3.11, the assumption in Lemma 3.12 entails that for each v_i there exists an l such that $\{\} \vdash v_i : V_i^l$, because for each v_i, we have $v_i \in \boldsymbol{R}_{V_i}$ (by assumption). The $\boldsymbol{R}$-Substitution property (from Lemma 3.12) entails the Strong Normalization Theorem (stated in Theorem 3.13) by using the empty context for the typing judgment.

Theorem 3.13 (Strong Normalization). *If $\{\} \vdash t : T^n$, then t terminates.*

4 Applications of Bounded Time λ-Calculus

This section elaborates further on the application of the BTC and SBT analysis to the motivating examples—timing channel analysis and functional hardware description (resp., Sections 4.1 and 4.2)—introduced earlier in Section 1. The presentation in this section is at a high-level and we leave the full development of these case studies to future work. However, sufficient detail is presented to illustrate the relevance of SBT analysis and its formalization in the BTC type system to these application domains.

4.1 Timing Channel Analysis

As mentioned in the Introduction, the execution time of a program may leak confidential information about inputs if the program takes different time to compute for different inputs. If the timing behavior of a program leaks information, then we say that the program has a *timing channel*. To rule out timing leaks, we have to ensure that the execution time of a program is independent of confidential inputs. In type-based approaches to timing channel analysis (see, for example, [36, 37]), the typing discipline guarantees that well-typed programs take the same amount of time to compute regardless of inputs. We describe here how the BTC framework can be used to achieve similar goals.

We consider the same BTC syntax (including Terms and Values), types, and reduction relation $\rightsquigarrow$. We identify the set of programs (with inputs) in BTC as a subset of the values:

Definition 4.1 (Values)**.** The set *program* of BTC values is given by the following:

$$v, w \in \mathit{program} ::= \lambda\, \boldsymbol{x}\ T\ t$$

Intuitively the program $\lambda\, \boldsymbol{x}\ T\ t$ takes an input of the type T and computes t. Observe that since we have product types, programs with multiple inputs can also be modeled in the same framework.

We assume that the *time* taken by a program $f \equiv \lambda\, \boldsymbol{x}\ T\ t$ on an input u of type T is the number of steps that it takes for $\mathit{app}\ f\ u$ to evaluate to a value v. Observe that since the step relation is deterministic (see Theorem 2.21) and terminating (see Theorem 3.13), the time taken by f on u is well-defined. However, the type system as given in Definition 2.6 does not guarantee that the time take by f is independent of the inputs.

Example 4.2. Let $B \equiv () + ()$, $\texttt{false} \equiv \mathit{inr}\ \texttt{nil}\ ()$ and $\texttt{true} \equiv \mathit{inl}\ \texttt{nil}\ ()$. Given terms t_1, t_2, t_3 let $\mathit{ite}\ t_1\ t_2\ t_3 \equiv \mathit{case}\ t_1\ \lambda z\ ()\ t_2\ \lambda z\ ()\ t_3$. Intuitively B models the type Booleans, `false` and `true` model falsehood and truth respectively, and $\mathit{ite}\ t_1\ t_2\ t_3$ models the conditional expression.

Consider the program and_1 defined as:

$$\mathsf{and}_1 \equiv \lambda\, \boldsymbol{x}\ B{\times}B\ (\mathit{ite}\ \pi_1 x\ (\mathit{ite}\ \pi_2 x\ \texttt{true}\ \texttt{false})\ \texttt{false}).$$

The program and_1 takes as input a pair of Boolean values and outputs the conjunction of the components of the pair. It is easy to see that and_1 has a timing channel as $(\mathsf{and}_1\ (\mathit{pair}\ \texttt{true}\ \texttt{true}))$ takes 7 evaluation steps to compute to a value and $(\mathsf{and}_1\ (\mathit{pair}\ \texttt{false}\ \texttt{true}))$ takes 4 steps.

On the other hand, consider the program and_2 defined as:

$$\mathsf{and}_2 \equiv \lambda\, \boldsymbol{x}\ B{\times}B\ (\mathit{ite}\ \pi_1 x\ \pi_2 x\ \pi_1 x).$$

The program and_2 also computes the conjunction of the components of a pair of Boolean values. The program and_2 takes the same time to compute for all possible pairs of Boolean values.

Thus, for identifying BTC programs that do not leak timing information, we need to restrict the set of well-typed terms. The reason for the possibility of timing channels is the *case* expression which permits different timings for the different *alternatives* of the case expression. Modifying the typing rule for the *case* expression to further require the case alternatives to have the same "execution time" ensures that well-typed programs are free of timing channels. Let us consider the typing system $\vdash_{\mathsf{tc}}$ for BTC programs, whose typing rules are the same as type inference system in Definition 2.6 with one exception. The typing rule Case of Definition 2.6 is replaced by the typing rule:

$$\frac{\Gamma \vdash_{\mathsf{tc}} s : (T+U)^n \qquad \Gamma \vdash_{\mathsf{tc}} t : (T \xrightarrow{l} S)^m \qquad \Gamma \vdash_{\mathsf{tc}} t : (U \xrightarrow{r} S)^p \qquad l+m=r+p}{\Gamma \vdash_{\mathsf{tc}} \mathit{case}\ s\ t\ u : S^{(n+l+m+2)}}\ (\textsc{Case})$$

Please see Fig. 1 for the formal definition of $\vdash_{\mathsf{tc}}$.

All theorems discussed in Section 2 and Section 3, stated with the typing relation $\vdash$ replaced by $\vdash_{\mathsf{tc}}$, continue to hold. In fact, we get a stronger version of type preservation:

Theorem 4.3 (Preservation for $\vdash_{\mathsf{tc}}$)**.** *If* $\{\} \vdash_{\mathsf{tc}} t : T^n$ *and* $t \rightsquigarrow u$ *then* $n > 0$ *and* $\{\} \vdash_{\mathsf{tc}} u : T^{n-1}$.

The variant of preservation stated in Corollary 3.6 takes the form:

Corollary 4.4. *If* $\{\} \vdash_{\mathsf{tc}} t : T^n$ *and* $t \overset{m}{\rightsquigarrow} u$*, then* $m \leq n$ *and* $\{\} \vdash_{\mathsf{tc}} u : T^{n-m}$.

Well-typed programs (with respect to $\vdash_{\mathsf{tc}}$) are free of timing channels:

Theorem 4.5 (Timing Channel Freedom)**.** *Let* f *be a program and* u_1, u_2 *be ground terms such that* $\{\} \vdash_{\mathsf{tc}} f : (T \xrightarrow{n} U)^0$, $\{\} \vdash_{\mathsf{tc}} u_1 : T^m$, $\{\} \vdash_{\mathsf{tc}} u_2 : T^m$*. Then for values* v_1, v_2 *and natural numbers* r_1, r_2 *such that* $\mathit{app}\ f\ u_1 \overset{r_1}{\rightsquigarrow} v_1$ *and* $\mathit{app}\ f\ u_2 \overset{r_2}{\rightsquigarrow} v_2$*, it must be the case that* $r_1 = r_2 = n + m + 1$.

Proof. By definition of $\vdash_{\mathsf{tc}}$, we have that $\{\} \vdash_{\mathsf{tc}} \mathit{app}\ f\ u_1 : U^{n+m+1}$ and $\{\} \vdash_{\mathsf{tc}} \mathit{app}\ f\ u_2 : U^{n+m+1}$. Thanks to Corollary 4.4, we have that $r_1, r_2 \leq n + m + 1$, $\{\} \vdash_{\mathsf{tc}} v_1 : U^{n+m+1-r_1}$ and $\{\} \vdash_{\mathsf{tc}} v_2 : U^{n+m+1-r_2}$. Since v_1, v_2 are values, the variant of Theorem 2.19, for $\vdash_{\mathsf{tc}}$ implies that $n + m + 1 - r_1 = n + m + 1 - r_2 = 0$ as desired. □

$$\frac{}{\Gamma, \boldsymbol{x}:T \vdash_{\mathrm{tc}} \boldsymbol{x}:T^0}\ (\textsc{Var}) \qquad \frac{\Gamma, \boldsymbol{x}:T \vdash_{\mathrm{tc}} t:U^n}{\Gamma \vdash_{\mathrm{tc}} \lambda \boldsymbol{x}\ T\ t:(T \xrightarrow{n} U)^0}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash_{\mathrm{tc}} f:(T \xrightarrow{n} U)^m\ \Gamma \vdash_{\mathrm{tc}} t:T^p}{\Gamma \vdash_{\mathrm{tc}} app\ f\ t:U^{(n+m+p+1)}}\ (\textsc{App}) \qquad \frac{}{\Gamma \vdash_{\mathrm{tc}} \mathtt{nil}:()^0}\ (\textsc{Nil})$$

$$\frac{\Gamma \vdash_{\mathrm{tc}} t:T^n}{\Gamma \vdash_{\mathrm{tc}} inl\ t\ U:(T+U)^n}\ (\textsc{Inl}) \qquad \frac{\Gamma \vdash_{\mathrm{tc}} u:U^n}{\Gamma \vdash_{\mathrm{tc}} inr\ u\ T:(T+U)^n}\ (\textsc{Inr})$$

$$\frac{\Gamma \vdash_{\mathrm{tc}} t:T^n\ \Gamma \vdash_{\mathrm{tc}} u:U^m}{\Gamma \vdash_{\mathrm{tc}} pair\ t\ u:(T \times U)^{n+m}}\ (\textsc{Pair})$$

$$\frac{\Gamma \vdash_{\mathrm{tc}} t:(T \times U)^n}{\Gamma \vdash_{\mathrm{tc}} \pi_1\ t:T^{(n+1)}}\ (\textsc{Pi}_1) \qquad \frac{\Gamma \vdash_{\mathrm{tc}} t:(T \times U)^n}{\Gamma \vdash_{\mathrm{tc}} \pi_2\ t:U^{(n+1)}}\ (\textsc{Pi}_2)$$

$$\frac{\Gamma \vdash_{\mathrm{tc}} s:(T+U)^n \quad \Gamma \vdash_{\mathrm{tc}} t:(T \xrightarrow{l} S)^m \quad \Gamma \vdash_{\mathrm{tc}} t:(U \xrightarrow{r} S)^p \quad l+m=r+p}{\Gamma \vdash_{\mathrm{tc}} case\ s\ t\ u:S^{(n+l+m+2)}}\ (\textsc{Case})$$

Figure 1. Type Inference System for Timing Channel Freedom

Example 4.6. Consider the program and_1 from Example 4.2 again. We can easily show that there is no type T and natural number n such that $\{\} \vdash_{\mathrm{tc}} \mathsf{and}_1 : T^n$. Essentially, this holds because the alternatives in the outermost *case* expression of and_1 take different times to compute. On the other hand, we can show that $\{\} \vdash_{\mathrm{tc}} \mathsf{and}_2 : (B \times B \xrightarrow{4} B)^0$ establishing that and_2 is free of timing channels.

4.2 Functional Hardware Description Languages

Functional languages have long been viewed as an appropriate organizing principle for hardware description [35]. One motivation for pursuing a functional language approach to hardware is to transfer the strengths of functional programming—i.e., its abstractions with their attendant software engineering and verification support—to hardware construction. Another motivation is that there is an intuitive correspondence between hardware circuitry and functional programs; e.g., a combinational circuit seems essentially functional because its outputs depend only on its inputs. But this correspondence, while useful, only goes so far, because the models of computation underlying functional languages and hardware contain some fundamental mismatches. The BTC type system provides a means for statically detecting one of these mismatches—specifically the failure of strongly-bounded termination—as we explain below for both combinational and sequential circuit designs.

A functional hardware description language is a domain-specific language for designing and implementing hardware circuits (Fig. 2, left) that is defined in terms of an expressive *Host* language (e.g., Haskell [4, 13, 17, 29] or Scala [5]). Its programs are then compiled (semi-)automatically into a *Target* hardware description language (e.g., VHDL or Verilog). What are the limits on the expressiveness of the embedded DSL in Fig. 2 (left)? That is, precisely which *Host* programs can be compiled faithfully to synthesizable *Target* designs? The question is non-trivial because hardware's finite storage capacity cannot accommodate unbounded data and control necessary to compile arbitrary *Host* programs. This is one of the aforementioned mismatches that could be detected by an adaptation of a BTC-like system to the *Host*'s type system.

One would like to have a precise answer to this hardware-synthesizability question; e.g., which Haskell programs can be represented faithfully as synthesizable VHDL/Verilog designs? Or, to put it another way, which Haskell programs may be considered as embedded FHDL programs and which ones cannot?

For combinational circuitry, a pure function (e.g., a Haskell function $f :: i \to o$), does not necessarily correspond to a combinational circuit because unboundedness either in data (e.g., the sizes of i and o) or in control (e.g., non-termination of f) cannot be accommodated within hardware's fixed storage capacity. Termination on all inputs is clearly a necessary condition for f's correspondence to a combinational circuit, but termination is, as we explain below, not sufficient. The sufficient condition is that f terminate on all inputs within n steps, for some fixed $n \in \mathbb{N}$, specified as $f :: i \xrightarrow{n} o$ in a BTC-like extension to Haskell's type system.

It is common practice [22] to portray sequential hardware designs as Mealy machines (Fig. 2, right) and the Mealy machine was originally conceived as a model of hardware synthesizability [24]. This sequential device takes two inputs on each clock cycle, external inputs $\mathbf{i}$ and internal state feedback from storage $\mathbf{s}$. Based on these inputs, combinational logic—marked "output and next state logic" in Fig. 2 (right)—computes the external output, $\mathbf{o}$, and the next state to be stored in storage. That the storage bank is connected to a clock is denoted in the diagram by a triangle (e.g., the "$\triangleleft$").

The compilation of function f to sequential hardware must, one way or another, extract a time slice function (call it $slice(f) :: (i, s) \to (o, s)$ in Haskell notation) to match the output and next-state logic of a Mealy machine. Extracting time slices may be due to explicit time partitioning by the programmer (e.g., with staging annotations [23] or resumption monads [29]) or be fully automated [13]. Either way, $slice(f)$ must exhibit strongly bounded termination because determining the clock speed of the Mealy machine depends on it.

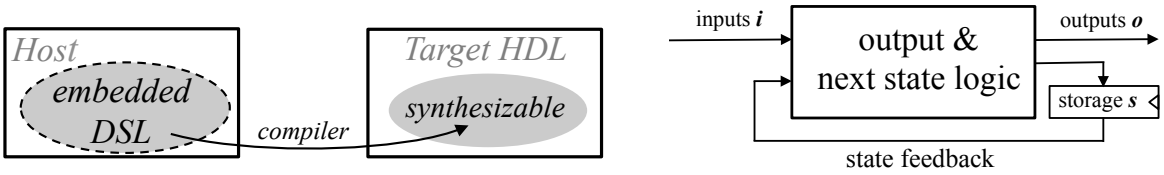

Figure 2. *Functional hardware description languages are DSLs embedded in a functional Host language (left). Hardware designs are traditionally portrayed as Mealy machines (right).*

5 Future Work and Conclusions

This work presented the bounded-time λ-calculus, BTC, and its formalization in Coq. The BTC type system enforces strongly-bounded termination—i.e., termination which is bounded in the number of computational steps. While the heart of this paper is the explication of the mechanized semantics and metatheory of BTC, we have also indicated the examples from security and hardware synthesis that inspired its development. Furthermore, how BTC applies to these areas was explained (albeit at a very high level). One natural next step for this research explores type inference and extensions to expressiveness (e.g., sub-typing). BTC was developed as an experimental offshoot of the formal semantic specification of the ReWire functional hardware description language [32] and it provides a formal basis for exploring the easing of recursion restrictions in ReWire. The authors expect to integrate a form of BTC (extended with suitable limited forms of recursion) with the aforementioned ReWire semantics. There is a theoretical question that motivated this work that we have alluded to in the previous section: what constitutes a hardware synthesizable functional program, and can the class of hardware-representable functions be identified (or, at least, well-approximated) by a suitable type system? While we do not pretend to have the answer to these questions, we suspect that BTC indicates a key component to any such answer (if, indeed, such an answer exists).

One reason to pursue a pure functional approach to synthesis [4, 13, 17, 29] is because pure functional languages lend themselves to formal verification and therein lies one important path forward for this work. If one wishes to verify the correctness of an HLS tool (i.e., *compiler* from Fig. 2, left), one must first have a precise definition of the embedded DSL, but this precision is missing in many of the aforementioned works. A type system that excludes non-synthesizable programs is, therefore, a necessary step to such a formal verification. We do not give a full account of synthesizability in this work, but rather describe the most surprising (although not in retrospect) part of this answer, having to do with strongly bounded termination behavior. There are other concerns that would of necessity be addressed in a synthesizability type system. For example, the possibility of representing data types as bit patterns (e.g., Diatchki et al [12] describe such an system as an extension of the Haskell type system).

References

[1] Andreas Abel. 2006. *A polymorphic lambda-calculus with sized higher-order types.* Ph.D. Dissertation. Ludwig-Maximilians-Universität München.

[2] Andreas Abel. 2008. Semi-continuous Sized Types and Termination. *Logical Methods in Computer Science* 4, 2 (2008), 1–33.

[3] Johan Agat. 2000. Transforming Out Timing Leaks. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 40–53.

[4] 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.

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

[6] Gilles Barthe, Benjamin Grégoire, and Colin Riba. 2008. Type-Based Termination with Sized Products. In *Computer Science Logic (LNCS)*, Vol. 5213. 493–507.

[7] Barthe, G., Frade, M. J., Giměnez, E., Pinto, L., and Uustalu, T. 2004. Type-based termination of recursive definitions. *Math. Struct. in CS* 14, 1 (2004), 97–141.

[8] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A core quantitative coeffect calculus. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems*, Vol. 8410. 351–370.

[9] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *44th ACM SIGPLAN POPL*. 316–329.

[10] Thierry Coquand. 1994. Infinite objects in type theory. In *Types for Proofs and Programs*. 62–78.

[11] Nachum Dershowitz and Zohar Manna. 1979. Proving termination with multiset orderings. In *Automata, Languages and Programming*. 188–202.

[12] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. 2005. High-level Views on Low-level Representations. In *10th ACM SIGPLAN ICFP*. 168–179.

[13] 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.

[14] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh. 2018. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 1583–1600.

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

[16] Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems*, Vol. 8410. 331–350.

[17] Andy Gill, Tristan Bull, Andrew Farmer, Garrin Kimmell, and Ed Komp. 2012. Types and Associated Type Families for Hardware Simulation

and Synthesis. *Higher Order Symbol. Comput.* 25, 2-4 (Dec. 2012), 255–274.

[18] Eduardo Giménez. 1998. Structural recursive definitions in type theory. In *Automata, Languages and Programming*, Kim G. Larsen, Sven Skyum, and Glynn Winskel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–408.

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

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

[21] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 410–423.

[22] R.H. Katz and G. Borriello. 2005. *Contemporary Logic Design*. Pearson Prentice Hall.

[23] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. 2004. A Methodology for Generating Verified Combinatorial Circuits. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT '04)*. 249–258.

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

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

[26] Nax Paul Mendler. 1991. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied logic* 51, 1-2 (1991), 159–172.

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

[28] Lars Pareto. 2000. *Types for Crash Prevention*. Ph.D. Dissertation. Chalmers University of Technology.

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

[30] Brian Reistad and David K. Gifford. 1994. Static Dependent Costs for Estimating Execution Time. *SIGPLAN Lisp Pointers* VII, 3 (July 1994), 65–78.

[31] Thomas Reynolds. 2020. Bounded Time Calculus Coq Codebase. Available from https://www.dropbox.com/s/l3f2nhghxkooxjh/tyde20-code.tar.gz?dl=0.

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

[33] Jorge Sacchini. 2011. *On type-based termination and dependent pattern matching in the calculus of inductive constructions*. Ph.D. Dissertation. École Nationale Supérieure des Mines de Paris.

[34] Jorge Luis Sacchini. 2014. Linear Sized Types in the Calculus of Constructions. In *12th Inter. Symp. on Functional and Logic Programming (FLOPS)*. 169–185.

[35] M. Sheeran. 1984. muFP, a Language for VLSI Design. In *ACM Symposium on LISP and Functional Programming*. 104–112.

[36] G. Smith. 2001. A new type system for secure information flow. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001*. 115–125.

[37] G. Smith and D. Volpano. 1998. Secure Information Flow in a Multi-threaded Imperative Language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. 355–364.

[38] W. W. Tait. 1967. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212.

[39] Hongwei Xi. 2002. Dependent Types for Program Termination Verification. *Higher Order Symbolic Computation* 15, 1 (March 2002), 91–131.

[40] D. Zhang, A. Askarov, and A. C. Myers. 2012. Language-based Control and Mitigation of Timing Channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 99–110.