Guarded Computational Type Theory

Jonathan Sterling
Computer Science Department
Carnegie Mellon University
jmsterli@cs.cmu.edu

Robert Harper
Computer Science Department
Carnegie Mellon University
rwh@cs.cmu.edu

Abstract

Nakano’s later modality can be used to specify and define recursive functions which are causal or synchronous; in concert with a notion of clock variable, it is possible to also capture the broader class of productive (co)programs. Until now, it has been difficult to combine these constructs with dependent types in a way that preserves the operational meaning of type theory and admits a hierarchy of universes $U_i$.

We present an operational account of guarded dependent type theory with clocks called $\text{CTT}_\circ$, featuring a novel clock intersection connective $\{k \div c1k\} \to A$ that enjoys the clock irrelevance principle, as well as a predicative hierarchy of universes $U_i$ which does not require any indexing in clock contexts. $\text{CTT}_\circ$ is simultaneously a programming language with a rich specification logic, as well as a computational metalanguage that can be used to develop semantics of other languages and logics.

CCS Concepts

• Theory of computation → Type theory; Modal and temporal logics;

Keywords

guarded recursion, clocks, type theory, operational semantics, dependent types

ACM Reference Format:

1 Introduction

In a functional programming language, every definable function is continuous in the following sense: each finite quantity of output is induced by some finite quantity of input. To make this more precise, if we consider the case of stream transformers $F : S \to S$, we can see that finite prefixes of the output depend only on finite prefixes of the input:

$$\forall \alpha : S. \forall \beta : S. \alpha \equiv_\beta \Rightarrow F(\alpha) \equiv F(\beta) \quad \text{(1)}$$

From a programming perspective, this can be rephrased in terms of reads and writes: for each write, the program is permitted to perform a finite but unbounded number of reads.

Causality

Another possible class of functionals are the ones that can be implemented by a program which performs at most one read for every write. These are called the causal functionals, and in the case of stream transformers, they are characterized by the following causality principle:

$$\forall \alpha : S. \forall \beta : S. \alpha \equiv_\beta \Rightarrow F(\alpha) \equiv F(\beta) \quad \text{(2)}$$

In other words, causal programs are the ones whose reads and writes proceed in lock-step. While we can surely carve out this class of functionals using predicates like (2) above, it is actually possible to define a new notion of stream $S_\triangleright$ such that all functionals $F : S_\triangleright \to S_\triangleright$ are automatically causal in the sense of (2). This kind of stream is called a “guarded stream”, and we will use the term “sequence” to refer to ordinary streams.

Whereas ordinary streams or sequences are usually formed as the greatest solution to the isomorphism $S \equiv \mathbb{N} \times S$, the guarded streams are formed using a special “later modality” $\triangleright$ due to Nakano, solving the isomorphism $S_\triangleright \equiv \mathbb{N} \times S_\triangleright$. Modalities of this kind usually enjoy at least the following principles:

$$\triangleright (A \to B) \to ((\triangleright A \to \triangleright B))$$

$$\triangleright (A \to B) \to ((\triangleright A \to B))\to (\triangleright A \to A) \to A$$

The ratio of reads and writes specified in the type of a stream transformer can be modulated by adjusting the number of later modalities in the input and the output to the function.

Nakano’s modality in semantics

What is remarkable about Nakano’s later modality is that fixed points for functions $F : \triangleright A \to A$ always exist, without placing any restriction on $F$ (such as monotonicity or positivity). Applied within a type-theoretic metalanguage, then, the later modality induces solutions to recursive domain equations which are not set-theoretically interpretible, such as the following classic definition of semantic types for a programming language with mutable store [5, 9]:

$$\text{Type} \equiv \left(\text{Loc} \xrightarrow{\text{fin}} \triangleright \text{Type} \right) \to \mathcal{P}(\text{Val})$$

The later modality captures and internalizes the basic features of less abstract techniques like step-indexing, enabling more streamlined definitions and proofs that elide the bureaucratic performance of explicit indexing and monotonicity obligations. Today, modalities of this kind are of the essence for modern program logics like Iris [24].

Programming applications

The fact that functions $F : \triangleright A \to A$ always have fixed points has beneficial consequences for the practice of (total) functional programming on infinite data. In particular, clumsy syntactic guardedness conditions which ensure productivity (such as those used in Coq [34], Agda [29] and Idris [15]) can be

---

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 the author(s) 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.

LICS ‘18, July 9–12, 2018, Oxford, United Kingdom
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5583-4/18/07...
https://doi.org/10.1145/3209108.3209153

\(^1\)The notation $\triangleright$ was originally used in Nakano [28].
which drops every second element from a stream! To define this very common in the real world. Consider, for instance, the function

\[ e : \forall \kappa. A \Rightarrow (\exists \kappa'. \kappa' \not\in \text{FreeClocks}(A)) \]

\[ \Delta; \Gamma \vdash e : \forall \kappa. A \]

\[ \Delta; \Gamma \vdash \Delta \kappa; \Gamma \vdash e : \forall \kappa. A \]

\[ k' \in \Delta \]

\[ \Delta; \Gamma \vdash e[k'] : A(\kappa \leftrightarrow k') \]

\[ \Delta; \Gamma \vdash \text{pure}(e) : \text{Ext}_{\kappa} A \]

\[ \Delta; \Gamma \vdash f : \text{Ext}_{\kappa}(A \rightarrow B) \]

\[ \Delta; \Gamma \vdash \text{force}(e) : \forall \kappa. A \]

\[ \Delta; \Gamma \vdash f : \text{Ext}_{\kappa} A \rightarrow A \]

\( (\forall \kappa. A) \equiv A \quad (\kappa \not\in \text{FreeClocks}(A)) \)

\( \forall \kappa. A \times B \equiv (\forall \kappa. A) \times (\forall \kappa. B) \)

Figure 1. Selection of rules from Atkey and McBride [6].

replaced with type structure, enabling more compositional styles of programming.

However, the later modality is too restrictive to be used on its own, because it rules out the functions which are not causal; but acausal functions on infinite data are perfectly sensible, and are very common in the real world. Consider, for instance, the function which drops every second element from a stream! To define this function, one would need a way to delete the modality; but without suitable restrictions, such an elimination principle would trivialize the modality and render it useless.

To resolve this problem, Atkey and McBride have introduced a notion of abstract clock \( k \) to represent “time streams” together with universal quantification \( \forall k \) over clocks, replacing Nakano’s modality with a clock-indexed family of modalities \( \text{Ext}_{k} \) [6].

Defining the type of \( k \)-guarded streams as the solution to the equation \( S_{k} \equiv \mathbb{N} \times \text{Ext}_{k} S_{k} \), it is possible to define the acausal function that drops every other element of a stream, with type \( (\forall k. S_{k}) \rightarrow (\forall k. S_{k}) \). The reason that this is possible is that their calculus exhibits the isomorphism \( (\forall k. \text{Ext}_{k} A) \cong (\forall k. A) \), as well as a clock irrelevance principle: \( (\forall k. A) \equiv A \) assuming that \( k \) is not free in \( A \); we summarize the constructs of this calculus in Figure 1.

1.1 Dependent type theory and guarded recursion

It has been surprisingly difficult to cleanly extend the account of guarded recursion with clocks to a full-spectrum dependently typed programming language which enjoys any combination of the following properties:

1. **Computational canonicity:** any closed element of type \texttt{bool} computes to either \texttt{tt} or \texttt{ff}.
2. **Simple universes:** a single predicative and cumulative hierarchy of universes \( U_{i} \) closed under base types, dependent function types, dependent pair types, lower universes, later modalities and clock quantifiers.
3. **Clock irrelevance:** if \( k \) is not mentioned in \( A \) and \( A \) is a type, then \( \forall k. A \) is equal to \( A \).

However, a dependent type theory with support for guarded recursion and clocks is desirable for multiple reasons; here, we have focused on causality as a useful construct for developing types qua behavioral specifications on program behavior, but there is also the potential to use such a dependent type theory as a computational metalanguage for developing and proving the semantics of other languages and logics, vaporizing the highly-bureaucratic step-indexed Kripke Logical Relations which usually must be employed.

The latter perspective is elaborated in the context of guarded dependent type theory without clocks in Paviotti et al. [30] as well as Bizjak et al. [11], and we anticipate that the addition of clocks will enable further developments along these lines.

1.2 Guarded Computational Type Theory

We contribute a new extensional and behavioral dependent type theory \( \text{CTT}_{\odot} \) (pronounced “Guarded Computational Type Theory”) for guarded recursion and clocks in the Nuprl tradition [2], enjoying the following characteristics:

1. Operational semantics and an immediate canonicity result at base types.
2. A clock-indexed later modality \( \text{Ext}_{k} A \) which requires no special syntax for introduction or destruction.
3. A decomposition of the clock quantifier from Bizjak and Magelberg [14] into a parametric part \( (k : \text{clk}) \rightarrow A \) and a non-parametric part \( (k : \text{clk}) \rightarrow A \). The former is an intersection, and enjoys the crucial clock irrelevance principle; the latter is the cartesian product of a clock-indexed family of sets (right adjoint to weakening).
4. A guarded fixed point combinator which can be assigned the type \( (\text{Ext}_{k} A) \rightarrow A \).
5. A predicative hierarchy of universes \( U_{i} \) closed under all the connectives, free of indexing by clock contexts.

Our operational account and canonicity result (Theorem 7) means that \( \text{CTT}_{\odot} \) can be regarded simultaneously as a programming language with a rich specification logic, and as a computational metalanguage for developing operational and denotational semantics of other languages and logics.

**Coq formalization and synthetic approach**

Using the Coq proof assistant, we have formalized the fragment of our type theory that contains universes, dependent function and pair types, booleans, the later modality, and the two clock quantifiers (intersection and product); the full Coq development is available in Sterling and Harper [33]. Throughout this paper, theorems and rules will be related to their Coq analogues using a reference like \text{Module}. \text{theorem name}.

We have used Coq’s type theory as a proxy for the internal language of the presheaf topos that we develop herein, axiomatizing in Coq whatever objects and principles come not from the standard type theoretic constructions, but are instead imported into the system via forcing. The entire construction of \( \text{CTT}_{\odot} \), then, is carried out within the internal language of the topos, an anti-bureaucratic measure which has made an otherwise daunting formalization effort feasible.

The idea of developing operational models of programming languages within the internal language of a topos is not new; see for instance Staton [32], Bizjak et al. [11] and Paviotti et al. [30]. However, we believe that ours is the first instance of this technique being
applied toward the development of semantics for a full-spectrum dependent type theory.

2 Programming in CTT

Following the computational meaning-theoretic tradition initiated by Martin-Löf [26], and developed further in the Nuprl project [2], we build Guarded Computational Type theory (CTT) on the basis of an untyped programming language, whose syntax is summarized in Figure 2.

In this paper, we distinguish between the syntax of formal terms and the language of programs; formal terms are used by clients of a formalism for type theory, whereas programs are the things which are actually endowed with operational meaning. For many languages, the difference between formal terms and programs is not so great, but for us the difference is essential; to avoid confusion, we distinguish between these levels using colors.

Formal Terms The grammar includes operators for both terms and types, which are not distinguished syntactically in any way. 

Typehood, equality and type membership are λ-calculus, but in order to simplify our metatheorems we have provided a primitive fixed point operator \( \text{fix} x \in M \).

The meaning explanation is, then, lifted to the formalism along an erasure map \( \| - \| \) that removes these decorations.

A similar, but more elaborate transformation of syntax (from formal terms to programs) is used here to facilitate the meaning explanation for guarded type theory in terms of the settings of a collection of clocks. During the verification of a program specification, the value of a clock may change (for instance, underneath the later modality), the most direct way to express this is to explicitly formulate the meaning explanation using a Kripke or presheaf-style semantics: a “possible world” consists of a collection of clocks and their settings, and we require specifications to account for the expansion of the world with new clocks and the alterations of their settings.

Doing so tends to clutter the meaning explanation by distributing the conditioning on clocks throughout the semantics, and disrupts a basic principle of type theory in the Martin-Löf tradition, which is that types should do little more than internalize the structures which are already present in the judgmental base.

An alternative, which we adopt here, is to formulate the semantics in a presheaf topos \( S_\circ \) which accounts all at once for clocks and the passage of time, so that the specifications given by types are implicitly conditioned on them. This conditioning, which is implicit when viewed from inside the topos, can be externalized and made explicit using the Kripke-Joyal forcing semantics of \( S_\circ \) [25].

To ensure that programs evolve appropriately along the transitions between clock worlds simultaneously with their specifications, we introduce a kind of “higher-order abstract syntax” which links clocks in programs directly to their meaning in the presheaf topos, as elements of the presheaf of clocks \( \mathbb{K} : S_\circ \). The passage to this new kind of syntax at the interface between the formalism and the semantics is managed by an elaboration function \( \| - \| \).

3 Mathematical Meaning Explanation

In the type-theoretic tradition of Martin-Löf, formal language is endowed with computational meaning through what is called a “meaning explanation”; this style of definition, which was first deployed by Martin-Löf in his seminal paper *Constructive Mathematics and Computer Programming* [26], is closely related to PER semantics and the method of computability. This computational perspective was developed to its fullest extent in Nuprl’s CTT [2, 18], which adds a theory of computational congruence to the picture, together with many new connectives including intersections, unions, subset comprehensions, quotients and image types.

A meaning explanation provides a semantics for types as specifications of the execution behavior of untyped programs. As such, the judgments of type theory express the compliance of a program with a specification, which can be of arbitrary quantifier complexity, and will not generally be decidable. Any implementation of type theory involves, in one form or another, a formal system for deriving correct judgments that is, by definition, recursively enumerable and often decidable.

To achieve various properties that are desirable of a formal system (sometimes including decidability), programs are often decorated with type information that is not needed during execution. The meaning explanation is, then, lifted to the formalism along an erasure map \( \| - \| \) that removes these decorations.

A similar, but more elaborate transformation of syntax (from formal terms to programs) is used here to facilitate the meaning explanation for guarded type theory in terms of the settings of a collection of clocks. During the verification of a program specification, the value of a clock may change (for instance, underneath the later modality), the most direct way to express this is to explicitly formulate the meaning explanation using a Kripke or presheaf-style semantics: a “possible world” consists of a collection of clocks and their settings, and we require specifications to account for the expansion of the world with new clocks and the alterations of their settings.

Doing so tends to clutter the meaning explanation by distributing the conditioning on clocks throughout the semantics, and disrupts a basic principle of type theory in the Martin-Löf tradition, which is that types should do little more than internalize the structures which are already present in the judgmental base.

An alternative, which we adopt here, is to formulate the semantics in a presheaf topos \( S_\circ \) which accounts all at once for clocks and the passage of time, so that the specifications given by types are implicitly conditioned on them. This conditioning, which is implicit when viewed from inside the topos, can be externalized and made explicit using the Kripke-Joyal forcing semantics of \( S_\circ \) [25].

To ensure that programs evolve appropriately along the transitions between clock worlds simultaneously with their specifications, we introduce a kind of “higher-order abstract syntax” which links clocks in programs directly to their meaning in the presheaf topos, as elements of the presheaf of clocks \( \mathbb{K} : S_\circ \). The passage to this new kind of syntax at the interface between the formalism and the semantics is managed by an elaboration function \( \| - \| \).

3.1 The semantic universe \( S_\circ \)

We will develop our semantic universe as a presheaf topos called \( S_\circ \) over a category of clock contexts and clock context morphisms. We will require the following things to exist in \( S_\circ \):

1. An object \( \mathbb{K} : S_\circ \) of clock names.
2. A family of logical modalities \( \star_{\kappa} \phi \) for clock names \( \kappa : \mathbb{K} \) and predicates \( \phi \) in \( S_\circ \).
\[ \begin{align*}
\kappa &::= k \\
M, A &::= x | \lambda x. M | \lambda k. M | M N | M k | (M, N) | M.1 | M.2 \\
& \quad \text{(clocks)} \\
& \quad \text{fix x in M | ★ | tt | ff | if(M; N; O)} \\
& \quad \text{ze | sup(M) | ifze(M; N; x; O) | sup(M; x; N)} \\
& \quad \text{recode(M; x; y; z; N)} \\
& \quad (x : A) \rightarrow B | (x : A) \times B | \Pi(x : A) B | Eq_A(M; N) \\
& \quad (k : c1k) \rightarrow A | (k \div c1k) \rightarrow A | \triangleright_k A \\
& \quad \text{void | unit | bool | nat | \_1} \\
\Delta &::= - | \Delta, k \\
\Psi &::= - | \Psi, x \\
\Gamma &::= - | \Gamma, x : A \\
\end{align*} \]

(clock contexts)

(terms)

(variable contexts)

(typing contexts)

**Figure 2.** The syntax of formal terms in Guarded Computational Type Theory (\(\text{CTT}_\infty\)). Formal terms \(M\) are identified up to renamings of their bound variables; by convention, bound variables are always assumed fresh.

When we define \(S_\infty\), we will arrange for the following principles to hold in its internal logic:

\[
\forall \kappa : \mathbb{K}. \| Y^\kappa \| \Rightarrow \forall \kappa : \mathbb{K}. \phi(\kappa)
\]

We require one additional axiom to hold for any object \(Y : S_\infty\) which is total and inhabited in a sense analogous to the notion from Birkedal et al. [9]:

\[
\forall \kappa : \mathbb{K}. \forall \phi : \Omega Y^\kappa. (\exists y : Y. \phi(y)) \Rightarrow \exists y : Y. \kappa(y)
\]

To construct \(S_\infty\) as a topos of presheaves, first define \(\mathbb{F}_\infty\): Cat as the free category with strictly associative binary products generated by a single object; explicitly, objects of \(\mathbb{F}_\infty\) are \(U \equiv \bullet^n\) for \(n > 0\). A map \(f : \bullet^n \rightarrow \bullet^m\) is a vector of projections, but can dually be regarded as a function between finite sets \(\mathbb{N}_m \rightarrow \mathbb{N}_n\).

Observe that the opposite category \(\mathbb{F}_\infty^+\) is a skeleton of the category of non-empty finite sets and all functions between them. \(\mathbb{F}_\infty\) is also a full subcategory of \(\mathbb{F}_\infty^+\): Cat, the free strict cartesian category generated by a single object (whose opposite is likewise a skeleton of the category of finite sets and all maps between them).

Define the presheaf of clock names \(N : \mathbb{F}_\infty^+\) as the representable functor \(y(\bullet^1)\). Next, define a functor \(\bullet[-] : \mathbb{F}_\infty \rightarrow \text{Pos}\) (with \(\text{Pos}\) the category of partially ordered sets) which will interpret assignments of times to clock names:

\[
\begin{align*}
\bullet[-] : \mathbb{F}_\infty \rightarrow \text{Pos} \\
\bullet[U : \mathbb{F}_\infty] &\equiv \omega^{N(U)} \\
\bullet[f : V \rightarrow U](\partial_V : \omega^{N(V)}) &\equiv (x : N(U)) \mapsto \partial_V(f^x) \\
\end{align*}
\]

Thinking of elements of \(\mathbb{F}_\infty^+\) as signifying finite and non-empty cardinalities of clock names, the action of \(\bullet[-]\) on objects takes such a cardinality \(U : \mathbb{F}_\infty^+\) to the \(U\)-fold product of the poset \(\omega\), ordered pointwise: in other words, it assigns the amount of “time left” to each clock.

Finally, using the covariant Grothendieck construction [20] we can build the total category \(\bullet : \text{Cat} \xrightarrow{\infty} \mathbb{F}_\infty\) in the following way. Objects are pairs \((U : \mathbb{F}_\infty^+, \partial_U : \bullet[U])\), i.e. collections of clock names together with an assignment; morphisms \(f : (V, \partial_V) \rightarrow (U, \partial_U)\) are \(\mathbb{F}_\infty^+\)-morphisms \(f : V \rightarrow U\) such that \(\partial[f] \leq \partial_U\) in \(\infty[U]\).

At this time it will be helpful to impose some notation: we will write \(\ell : \emptyset \rightarrow \mathbb{F}_\infty\) for the induced projection functor, and we will use boldface letters \(U, V\) to range over objects \((U, \partial_U), (V, \partial_V) : \emptyset\).

**The semantic universe \(\mathbb{S}_\infty\)** Finally, we define our semantic universe as the presheaf topos \(\mathbb{S}_\infty = \mathbb{S}_\infty\). This “topos of clocks” defined above inherits a rich internal logic which corresponds to a combination of cartesian/structural nominal logic\(^1\) and guarded recursion.

The topos \(\mathbb{S}_\infty\) is related to the models considered by Bizjak and Møgelberg [13], except that rather than constructing a family of presheaf toposes fibered over clock contexts, we combine clock contexts with time assignments into a single base category, and take the topos of presheaves over that; our topos is nearly identical to the presheaf category considered independently in Bizjak and Møgelberg [14].

One minor difference between our model and those of Bizjak and Møgelberg is that in order to close the internal logic of \(\mathbb{S}_\infty\) under the clock irrelevance axiom described above, we decided to rule out empty clock contexts; this condition is equivalent to taking a sheaf subtopos of the presheaves over all clock contexts.

**The object of clock names** We need to exhibit an object in the presheaf topos \(\mathbb{S}_\infty\) whose elements are the “available” clock names (without regard to their time assignments). First observe that the representable object \(N\) plays exactly this role in the category \(\mathbb{F}_\infty^+\): at clock context \(\bullet^N\) it consists in the set of morphisms \(\bullet^N \rightarrow \bullet^1\), which has cardinality \(n\). However, this object resides in the wrong topos, since we need to define an object \(\mathbb{K} : \mathbb{S}_\infty\). To achieve this, we use the reindexing functor \(\ell^* : \mathbb{F}_\infty^+ \rightarrow \mathbb{S}_\infty\) induced by precomposing the projection \(\ell : \mathbb{S}_\infty \rightarrow \mathbb{F}_\infty^+, \text{defining } \mathbb{K} \equiv \ell^*N\).

**Notations and morphisms** We write \(U[\kappa \mapsto n]\) to mean \((U, \partial_U[\kappa \mapsto n])\), where \(\partial_U[\kappa \mapsto n]\) means the adjustment to \(\partial_U\) which replaces \(\partial_U(\kappa)\) with \(n\). Finally, for the map that increments the time assigned to a clock, we write \(\kappa[+1] = \lambda : U \rightarrow U[\kappa \mapsto \partial_U(\kappa) + 1]\).

**Defining the \(\bullet_k\) modalities** We define the \(\bullet_k\) modalities by their forcing clause in the Kripke-Joyal semantics of \(\mathbb{S}_\infty\):\(^5\)

\[
\begin{align*}
U \models \bullet_k \phi(\alpha) &\iff \\
&\begin{cases} 
T & \text{if } \partial_U(\kappa) \equiv 0 \\
U[\kappa \mapsto n] \models \phi(\kappa[+1]) & \text{if } \partial_U(\kappa) \equiv n + 1
\end{cases}
\end{align*}
\]

\(^5\text{That is, the logic of nominal substitution sets} [22, 32].\)

\(^1\text{Usually the forcing clauses should be taken as theorems rather than as definitions. However, in a Grothendieck topos, it is possible to define a subobject by its forcing clause: the result is well-defined when the definition is monotone (and also local, in the case of sheaf toposes).}\)
By a similar definition, it is possible to define an analogous operator in the internal type theory of $\mathcal{S}_\Diamond$, i.e. a fibered endofunctor $\mathbf{r} : \mathcal{S}_\Diamond/X \times \mathbb{X} \to \mathcal{S}_\Diamond/X \times \mathbb{X}$; however, we have only needed the logical modality in our construction.

All the other forcing clauses are completely standard; for a reference on Kripke-Joyal forcing, see Mac Lane and Moerdijk [25].

3.2 Programming language and operational semantics

In Section 2 (Figure 2) we gave a grammar for the formal terms of CTT$_\gamma$; however, in our semantics, we employ a second notion of syntax which is constructed as an inductive definition internal to $\mathcal{S}_\Diamond$; this is the language of programs, and differs from the syntax of formal terms in two respects:

1. Clocks in programs are imported directly from the metatheoretic object of clocks $\mathbb{X} : \mathcal{S}_\Diamond$; so the family of operators $\mathbf{r}_k$ is indexed in $\mathbb{X}$ exactly the same way that $U_i$ is indexed in $i : M$.

2. The binding of clocks (such as in the clock intersection operator) is represented using the exponential $\mathbb{X}^{\mathbb{X}} : \mathcal{S}_\Diamond \to \mathcal{S}_\Diamond$.

Remark 1 (Generalized Syntax). The idea of using the exponential of the metalanguage in the syntax of a programming language is not new. Infinitary notions of propositional syntax can be traced back as far as Brouwer’s $I$-inference in the justification of the Bar Principle [16], and have more recently been developed in Nuprl semantics [31], as well as in the context of higher-order focusing [36].

We will define the inductive family $\text{Prog}_n$ of programs with $n$ free variables in $\mathcal{S}_\Diamond$ using an internal inductive definition, summarized in Figure 3.

Our syntax forms a substitution algebra, and we write $\text{Var}_n \to \text{Prog}_n$ for the injection of variables into terms; we write $M \gamma : \text{Prog}_n^n$ for the action of the substitution $\gamma : \text{Prog}^n_{\text{var}}$ in $M : \text{Prog}_m^n$.

Internal operational semantics

Programs are endowed with operational meaning through the definition of a transition system, an illustrative fragment of which we present in Figure 3. This defines predicates $\mathbf{val} : \mathcal{P}(\text{Prog}_0)$ and $\mathbf{rel} : \mathcal{P}(\text{Prog}_0 \times \text{Prog}_0)$ in $\mathcal{S}_\Diamond$. Write $\mathcal{Val} : \text{S}_\Diamond$ for the subobject $(M : \text{Prog}_0 \vdash M \text{val})$.

Write $\mathbf{rel} \to \mathbf{rel}^*$ for the reflexive-transitive closure of $\mathbf{rel}$. We now define approximation and computational equivalence judgments $\mathbf{rel} \to \mathbf{rel}^*$ as $\mathcal{P}(\text{Prog}_0 \times \text{Prog}_0)$ respectively for closed programs as follows:

$$M_0 \approx M_1 \triangleq \forall M_2 : \mathcal{Val} : M_0 \mathbf{rel}^* M_2 \Rightarrow M_1 \mathbf{rel}^* M_2$$

$$M_0 \approx M_1 \triangleq M_0 \approx M_1 \land M_1 \approx M_0$$

The latter is extended to a computational equivalence judgment for open programs $\approx_n$ as $\mathcal{P}(\text{Prog}_n \times \text{Prog}_n)$ by quantifying over total substitutions.

$$M_0 \approx_n M_1 \triangleq \forall \gamma : \text{Prog}^n_0 : M_0 \gamma \approx M_1 \gamma$$

It would be desirable to extend this theory to a relation of computational congruence, as pioneered by Howe [23]; however, for our immediate purposes it has sufficed to require types only to respect the approximation relation defined above.

Definition 2 (Computational PERs). A partial equivalence relation is a binary relation which is both symmetric and transitive. Such a relation $\mathcal{R}$ on $\text{Prog}_0$ is called computational when it respects approximation in the following sense: if $(M_0, M_1) \in \mathcal{R}$ and $M_0 \approx M'_0$, then $(M'_0, M_1) \in \mathcal{R}$.

Telescopes

To capture the syntax of contexts and we define the inductive family $\mathcal{T}_n$ of telescopes of length $n$ as follows:

$$\mathcal{T}_0 : \mathcal{T}_n \quad \quad \Gamma : \mathcal{T}_n \quad A : \text{Prog}_n$$

$$\mathcal{T}_n$$

Elaborating terms

We now sketch the elaboration of the program terms of Section 2 into programs; approximately, a term $M$ with free formal clock variables $\Delta$ and free term variables $\Psi$ will be elaborated to a morphism $\| \Delta \| : \| \Psi \| : \mathbb{X}^{\mathbb{X}} \to \text{Prog}[\Psi]$.

Notation 3. When $\Psi$ is a list, we write $||\Psi||$ for its length, and we write $||\Psi[x]||$ for the index $i < ||\Psi||$ of the element $x$ in $\Psi$, presuming $\Psi \neq \varepsilon$.

We present here only a few of the most illustrative cases; the remainder of the elaboration can be found in technical report, and in our Coq formalization [33].

$$\| \Delta \| : \| \Psi \| : \mathbb{X}^{\mathbb{X}} \to \text{Prog}[\Psi]$$

Elaborating contexts

Next, we elaborate contexts $\Gamma$ with free formal clock variables $\Delta$ as morphisms $\| \Delta \| : \mathbb{X}^{\mathbb{X}} \to \mathcal{T}[\Gamma]$, writing $\pi(\Gamma)$ for the sequence $\bar{x}_i$ when $\Gamma \equiv \pi x_i : A_i$.

$$\| \Delta \| : \| \Psi \| : \mathbb{X}^{\mathbb{X}} \to \text{Prog}[\Psi]$$

To save space, we may write $\|M\|$ or $\|\Gamma\|$ for the elaboration of a term or a context respectively, when the parameters are obvious.

3.3 Full type system hierarchy

At a high level, a type system in the sense of Allen [3] is an object which distinguishes some programs as types, and specifies what programs will be the elements of those types, and when they will be considered equal. Writing rel(X) for $\mathcal{P}(X \times X)$, we define a candidate type system to be a relation $\tau : \mathcal{P}(\text{Prog}_\gamma \times \text{rel}(\text{Prog}_0))$ in $\mathcal{S}_\Diamond$. We will write $\mathcal{T}_{\text{cand}}$ for the collection of such candidate type systems, i.e. $\mathcal{T}_{\text{cand}} : \mathcal{S}_\Diamond \triangleq \mathcal{P}(\text{Prog}_0 \times \text{rel}(\text{Prog}_0))$.

Let us now define notation for some assertions about candidate type systems $\tau : \mathcal{T}_{\text{cand}}$:

$$\tau \vdash A \triangleq B \triangleq \exists A : \text{rel}(\text{Prog}_0). (A, A) \in \tau$$

$$\tau \vdash M_0 \triangleq M_1 \in A \triangleq \exists A : \text{rel}(\text{Prog}_0). (A, A) \in \tau$$

A candidate type system $\tau : \mathcal{T}_{\text{cand}}$ can have the following characteristics:

1. It is called extensional if it is the graph of a partial function $\text{Prog}_0 \to \text{rel}(\text{Prog}_0)$.

While this construction cannot be called “ordinary syntax”, it is an inductive definition that can be built up explicitly using the fact that $\mathcal{S}_\Diamond$ models indexed W-types [27].
Next, we briefly sketch the mean-
out in any topos, because the Knaster-
arski theorem guarantees
operator manually by taking the least fixed point of a monotone op-
positive signatures.

erately extend to the type systems that we consider in this paper,

c by Crary [19] and formalized by Anand and Rahli [4], is to use an
later modalities, clock intersection types and universes.

Next, we will show how to

3.4 Closure under type formers other than universes

Next, we will show how to close a candidate type system under
the type formers of

CTT, namely booleans, natural numbers,
dependent functions types, dependent pair types, equality types, later modalities, clock intersection types and universes.

The simplest way to carry out this construction, as pioneered
by Crary [19] and formalized by Anand and Rahli [4], is to use an
inductive definition of a closure operator \( \text{c}[-] : \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}} \)
on candidate type systems. However, this method does not imme-
diately extend to the type systems that we consider in this paper,
because it is not clear how to fit the clause for the later modality
into the usual schemata for inductive definitions based on strictly
positive signatures.

Therefore, as advocated by Allen [3], we will build up our closure
operator manually by taking the least fixed point of a monotone op-
erator on candidate type systems; this construction can be carried
out in any topos, because the Knaster-Tarski theorem guarantees
a least fixed point for any monotone operator on a complete lattice [21].

First, we define some notation for closing relations and type
systems under evaluation to canonical form:

\[
\mu : \text{rel}(\mathcal{T}_{\text{cand}}) \rightarrow \text{rel}(\mathcal{T}_{\text{cand}})
\]

\[
\lambda : \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}}
\]

\[
\tau : \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}}
\]

In Figure 4, for an initial candidate type system \( \sigma : \mathcal{T}_{\text{cand}} \), we def-
e an endomorphism on candidate type systems \( \text{cand} \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}} \)
which extends a type system with all the non-universe connectives
of \( \text{CTT}^\ast \).

Theorem 4 (Closure.Clo.monotonicity). For any candidate type
system \( \sigma : \mathcal{T}_{\text{cand}} \), the function \( \text{cand} \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}} \)
is monotone.

Proof. By case on the type closure clauses above, which are them-
selves each monotone.

\( \square \)

Corollary 5 (Closure.Clo.t,Closure.Clo рол). By the Knaster-
Tarski theorem, the function \( \text{cand} \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}} \)
takes the operator that takes
\( \sigma : \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}} \) for the fixed point \( \mu(\text{cand} \mathcal{T}_{\text{cand}}) \).

We will write \( \text{c}[-] : \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}} \) for the operator that takes
\( \sigma : \mathcal{T}_{\text{cand}} \rightarrow \mathcal{T}_{\text{cand}} \) to the fixed point \( \mu(\text{cand} \mathcal{T}_{\text{cand}}) \).

3.5 The full universe hierarchy

The next step in the construction is to build up the universe hier-
archy. Following Allen [3], we define the “spine” of the universe
hierarchy as a sequence of type systems \( \nu : \mathcal{T}_{\text{cand}}^\mathcal{H} \)
that contains at each level only types which evaluate to universes:

(3.5.1)

The sequence above is well-defined by complete induction on
the index. We are now equipped to define a new sequence of type
systems which is at each level closed under all the ordinary type
formers as well as smaller universes:

\( \tau \triangleq \text{c} \langle \nu \rangle \)
Finally, we can capture the entire countable hierarchy in a single
type system \( \tau_{\omega} \), which is the join of the entire sequence:

\[
\tau_{\omega} \triangleq \bigvee_{\omega \in \mathbb{N}} \tau_i
\]

When we explain the meaning of judgments, it will always be
done with respect to this maximal type system.

**Theorem 6** (\( \tau_{\omega} \) type system). *The ultimate candidate type system \( \tau_{\omega} \) is in fact a type system.*

### 3.6 Meaning explanation

In this section, we give a mathematical meaning explanation to the
formal judgments of \( \mathcal{J} \) in \( S_\Box \).

1. Functional equality of elements \( \Delta \mid \Gamma \gg \) \( M_0 \equiv M_1 \in A \) means that in clock context \( \Delta \) and variable context \( \Gamma, M_0 \) and \( M_1 \) are equal elements of type \( A \). This form of judgment requires that \( \Gamma, M_0, M_1, A \) mention only clocks from \( \Delta \), and that \( M_0, M_1, A \) mention only variables from \( \Gamma \).

2. Untyped open conversion \( \Delta \mid \Psi \vdash M_0 \equiv M_1 \) means that \( M_0 \) and \( M_1 \) are Kleene equivalent in all their instantiations. This form of judgment requires that \( M_0, M_1 \) mention only clocks from \( \Delta \) and variables from \( \Psi \).

**The meaning of judgments** We interpret each formal judgment
\( \mathcal{J} \) as a proposition \( [\mathcal{J}] : \Omega \) in \( S_\Box \):

\[
[(\Delta \mid \Gamma \gg M_0 \equiv M_1 \in A)] \triangleq \forall \rho : K[A].
\]

\[
\tau_{\omega} \vdash \llbracket \Gamma \rrbracket \llbracket \rho \rrbracket \text{ ctx}
\Rightarrow \tau_{\omega} \vdash \llbracket \Gamma \rrbracket \llbracket \rho \rrbracket \gg \llbracket A_0 \rrbracket \llbracket \rho \rrbracket \equiv \llbracket A_1 \rrbracket \llbracket \rho \rrbracket
\Rightarrow \tau_{\omega} \vdash \llbracket \Gamma \rrbracket \llbracket \rho \rrbracket \gg \llbracket M_0 \rrbracket \llbracket \rho \rrbracket \equiv \llbracket M_1 \rrbracket \llbracket \rho \rrbracket
\]

Observe that the usual presuppositions of the equality judgment
(context validity and type functionality) are taken as assumptions:
the principle can be summarized as "garbage in, garbage out". Du-
ally, we could have chosen to regard them as consequences, which
would lead to a slightly different collection of validated rules.

**Canonicity at base type** Define 2 : \( S_\Box \) for the boolean object in
our semantic framework, which has two global elements \( 2_0, 2_1 : 2 \).
Define an embedding \( \llbracket - \rrbracket_2 \) from this object into our formal term
language as follows:

\[
\llbracket 2_0 \rrbracket_2 = \text{tt} \quad \llbracket 2_1 \rrbracket_2 = \text{ff}
\]

Now we can state the canonicity theorem for \( \text{CTT}_\Box \).

**Theorem 7** (Canonicity). *For any closed expression \( M \) such that \( \llbracket \cdot \rrbracket \gg M \equiv \text{bool} \), there exists some \( b \in 2 \) such that \( \llbracket \cdot \rrbracket \gg M \equiv \llbracket b \rrbracket_2 \).*

**Corollary 8.** *The type theory \( \text{CTT}_\Box \) is consistent in the sense that
there is no inhabitant of void.*

### 3.7 Validated rules

In Figure 5, we present a small selection of the rules which we have
validated in our Coq formalization of \( \text{CTT}_\Box \); we omit most of the
standard rules of ordinary extensional type theory, which are also
valid in our semantics.

### 3.8 Examples: revisiting streams

Using these rules, we can derive some typing lemmas for guarded
streams and coinductive sequences of bits.

\[
\text{stream} \triangleq \lambda k. \text{fix } A \text{ in bool } \gg \lambda k \text{. A}
\]

\[
\text{sequence} \triangleq (k \times \text{clk}) \rightarrow \text{stream } k
\]

\[
\text{ones} \triangleq \text{fix } x \in (\text{tt}, x)
\]


### 4 Survey of Related Work

#### 4.1 Guarded Dependent Type Theory

The standard model of guarded recursion without clocks is the topos
\( \Omega \), the presheaves on the poset of natural numbers regarded
as a category [9]. This topos can be regarded as a denotational model
for a variant of Martin-Löf’s extensional type theory equipped with
the \( \gg \) modality. By indexing this topos over a category of clock contexts \( \Delta \), it is possible to develop a model of extensional type
theory with clock quantification called \( \text{GDTT} \) [12, 13]. In order to
justify a crucial *clock irrelevance* principle, it is necessary to index
universes in clock contexts, i.e. \( U_\Box \).

In the dependent setting, some difficulties arise when devising a
syntax for the semantic type theory of this indexed category.
A selection of the rules which we have proved correct for our type theory.

Figure 5. A selection of the rules which we have proved correct for our type theory.

order to make sense of the "delayed application" operator \( \otimes \) in the context of dependent function types, it was necessary to introduce a notion of _delayed substitution_ \( \xi \equiv [\xi \mapsto \vec{\tau}] \) which pervades the term language, introducing term formers like \( \text{s}^k \xi. A \) and next\( ^k \xi. e \). On the bright side, delayed application can be defined in terms of delayed substitution.

However, the equational theory for delayed substitutions is fairly sophisticated, and an operational (computational) interpretation of \text{GDTT} has not yet been proposed at the time this article was written; as such, a canonicity theorem for this system is still forthcoming.

4.2 Orthogonality and clock irrelevance

In a more recent development [14], a denotational model of \text{GDTT} has been developed that differs from that of Bizjak and Møgelberg [13] in a few crucial ways.

**Unified base category** The fibered topos presentation of the Bizjak and Møgelberg [13] work has been replaced with a presheaf topos over a single unified base category, discovered independently from the unified base category which we introduce in Section 3.1. Taking presheaves over this unified base category simplifies the model significantly, and also makes available the standard solution to the substitution coherence problem for (denotational) presheaf models of dependent type theory.\(^7\)

The proposed base category of Bizjak and Møgelberg [14] differs from ours mainly in that they allow empty worlds, whereas we restrict our base category to those worlds which contain at least a single clock.

**Orthogonality** Bizjak and Møgelberg define a presheaf of clocks \( C \) which is the same as our object of clocks \( \mathbb{K} \) which we introduce in Section 3.1; then, the clock quantifier is represented in the internal language of their presheaf topos as a dependent product over \( C \), i.e. \( \Pi_{\kappa:C} A(\chi) \).

Defined in this way, the clock quantifier cannot be a priori parametric with respect to clocks / time objects; therefore, in order to validate the clock irrelevance axiom, the authors have identified an orthogonality condition on objects, which in essence closes the internal language of the presheaf topos under just those types which are compatible with the irrelevance principle for the clock quantifier.

Unfortunately, the subtopos of time-orthogonal objects does not contain the standard Hofmann-Streicher universes, because

\(^7\)This is to use an alternative construction of the slice categories \( \tilde{\mathcal{C}}/X \), as the presheaves on the total category of \( X \).
which treat clocks parametrically; then, within this subcategory, we have defined two separate clock quantifiers which at the time of writing have a decidable typing result, nor a strong internal type theory of the topos of trees. Although GDTT does not currently support only a single clock, but it is plausible that virtue lies in pursuing the intrinsic path, especially as far as implementability are concerned. The calculus developed in Bahr et al. [7] (and more recently, the ideas contained in Clouston et al. [17]) are likely to provide the basis for a syntactic account of guarded recursion which is sound for our model, but closer to implementation.

4.4 Clocked Type Theory

Recently, an alternative to GDTT called CloTT called Clocked Type Theory (CloTT) has been proposed, which enjoys a computational interpretation with a canonicity result [7]; it is plausible that Clocked Type Theory shall have a decidable typing relation. Notably, Clocked Type Theory does not validate any clock irrelevance rule; the authors propose to address this in a cubical version of CloTT by adding a special path axiom which realizes this principle, by analogy with the technique used in GCTT to account for restricted unfoldings of fixed points. In the presence of this axiom, canonicity for CloTT can still be made to hold in the context which contains only a single clock.

Discussion Clocked Type Theory looks like a promising path toward a well-behaved intrinsic account of guarded recursion with clocks. In the present paper, our efforts have been focused exclusively on developing the behavioral account of guarded type theory in the style of Martin-Löf’s meaning explanation, in which programs can be regarded as existing separately from their types; here, general recursive programs can be written and shown to be (causal, productive, total) in a semantic sense, using the type theory as a program logic.

We perceive, however, that virtue lies in pursuing the intrinsic path, especially as far as implementability are concerned. The calculus developed in Bahr et al. [7] (and more recently, the ideas contained in Clouston et al. [17]) are likely to provide the basis for a syntactic account of guarded recursion which is sound for our model, but closer to implementation.

4.5 Sized Types and size quantifiers

Our decomposition of the quantifier $\forall k$ from GDTT into a parametric part $(k \cdot clk) \rightarrow A$ and a non-parametric part $(k : clk) \rightarrow A$ mirrors the state of affairs in the literature on sized types, which is another account of type-based guarded recursion [1].

5 Perspective and Future Work

We have developed and formalized a computational account of guarded dependent type theory with clocks, enjoying several desirable characteristics not found together in other existing models: computational canonicity, clock irrelevance and ordinary universes. We have made the following contributions toward a simpler, more computational account of guarded dependent type theory:

Implementation, proof theory, and syntax We have not yet tackled the project of developing an ergonomic proof theory for CTT which can be used to interact with the semantics presented here. The natural deduction style rules which we have given here are, while convenient for paper presentations, not what one would use in a serious implementation. To build a proof theory for CTT, we must negotiate new forms of judgment with decidable presupposition. Therefore, while we have indeed developed a programming language for guarded type theory with clocks that omits explicit syntax for delayed substitutions, this should be understood in terms of the conceptual order of semantics and proof theory which is endemic in computational type theory. In particular, while our programming language and type theory has no need for such a construct, in a proof language for CTT it would be necessary to account for the syntactic structure of the later modality’s elimination; we anticipate
We are thankful to Carlo Angiuli, Lars Birkedal, Aleš Bizjak, Jonas Frey, Daniel Gratzer, Adrien Guatto, Pieter Hofstra, Bas Spitters, Sam Staton, and Joseph Tassarotti for helpful discussions on the semantics of guarded recursion, clock names and universe hierarchies. Thanks to David Christiansen for his comments on a draft of this paper.

The authors gratefully acknowledge the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

Acknowledgments

References


