Guarded Computational Type Theory

JONATHAN STERLING, Carnegie Mellon University
ROBERT HARPER, Carnegie Mellon University

We develop a computational interpretation of guarded dependent type theory with clocks called CTT which enjoys a straightforward operational semantics and immediate canonicity result for base types. Our realizability-style presentation of guarded type theory is a computational and syntactic alternative to category-theoretic accounts of guarded recursion, emphasizing type theory’s role as the ultimate logic of programming.

CCS Concepts: •Theory of computation → Type theory;

ACM Reference format:
DOI: 10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION

What counts as an effective or computable transformation of infinitary data? Answers to this question distinguish different concrete notions of computation, including the radically intuitionistic and the classical. The discovery of the duality (observation, construction), with programming serving as its mediating unity, has had a clarifying effect on this question: we may simply ask, What amount of observation is licensed by an intended construction?

Continuity. In the case of intuitionistic computation, finitary constructions license (arbitrary) finitary observations. For a functional $F : N^N \rightarrow N^N$, which signifies an intention towards infinitely many finitary constructions, this slogan induces the following continuity principle, requiring that finite prefixes of the output depend only on finite prefixes of the input:

$$\forall \alpha : N^N. \forall i : N. \exists n : N. \forall \beta : N^N. \alpha =_n \beta \Rightarrow F(\alpha)_i \equiv F(\beta)_i$$

(continuity)

Discontinuity. Classical "computation" represents an extreme case, in which there is no connection between licensed observations and intended constructions: for instance, we can form a classical function which decides the extensional equality of two sequences, which is a finitary construction on the basis of an infinitary observation.

Causality. It is also possible to study another extreme case, where a finer relationship is imposed between construction and licensed observations, namely where constructions license observations of at most the same size: construction and observation proceed in lock-step. This kind of programming is called causal programming. A functional $F : N^N \rightarrow N^N$ is causal when the $i$th prefix of its output depends only on the $i$th prefix of its input:

$$\forall \alpha : N^N. \forall i : N. \forall \beta : N^N. \alpha =_i \beta \Rightarrow F(\alpha)_i \equiv F(\beta)_i$$

(causality)

1.1 Causality from type structure

Rather than viewing causality as an extrinsic feature of a functional on ordinary sequences, we wish to see how this characteristic can be induced by relaxing our notion of sequence; following the introduction of guarded
With the following desirable operations, which together form a toolkit for building (guarded) recursive programs:

1. Modality of the appropriate clock: \( \kappa \) variable in/finite data? Atkey and McBride (2013) demonstrated how to do exactly this, by introducing a notion of

Can we form a type of sequences for which it is possible to write anti-causal, but still continuous, functions on not causal (to define it, we would need to be able to eliminate the \[ \] term language, introducing term formers like \( \text{delayed substitution} \). In order to make sense of the “delayed application” operator \( \odot \) in the context of dependent function types, it was necessary to introduce a notion of \( \text{delayed substitution} \) \( \xi \equiv [\bar{\xi} \leftarrow \bar{e}] \) which pervades the term language, introducing term formers like \( \triangleright^\kappa \). On the bright side, delayed application can be defined in terms of delayed substitution.

It is easy to see that whilst we can construct a program to increment every element of a stream (see below), there is no way to define a program which drops every second element of a stream, because such a function is not causal (to define it, we would need to be able to eliminate the \( \triangleright \) modality).

\[
\text{incr : stream} \rightarrow \text{stream} \\
\text{incr} \triangleq \text{fix}(\lambda F. \lambda \alpha. (\text{fst}(\alpha) + 1, F \odot \text{snd}(\alpha)))
\]

### 1.2 Relaxing causality with clocks

Can we form a type of sequences for which it is possible to write anti-causal, but still continuous, functions on infinite data? Atkey and McBride (2013) demonstrated how to do exactly this, by introducing a notion of clock variable \( \kappa \), replacing \( \triangleright \) with \( \triangleright^\kappa \), and adding a new clock quantifier \( \forall \kappa \), under which it is possible to drop a later modality of the appropriate clock:

\[
\begin{align*}
\Delta, \kappa; \Gamma \vdash e : A & \quad \Delta; \Gamma \vdash : \forall \kappa. A & \quad \kappa' \in \Delta & \quad \kappa' \notin \text{FreeClocks}(A) \quad \Delta; \Gamma \vdash : \forall \kappa. \triangleright_\kappa A \\
\Delta; \Gamma \vdash \lambda \alpha. e : \forall \kappa. A & \quad \Delta; \Gamma \vdash e[\kappa'] : A[\kappa \leftarrow \kappa'] & & \Delta; \Gamma \vdash \text{force}(e) : \forall \kappa. A \\
\forall \kappa. A \equiv A & \quad (\kappa \notin \text{FreeClocks}(A)) & \quad \forall \kappa. A \times B \equiv (\forall \kappa. A) \times (\forall \kappa. B)
\end{align*}
\]

For our example, streams are defined in much the same way as before, as the solution to the equation \( \text{stream}[\kappa] \equiv \mathbb{N} \times \triangleright^\kappa \text{stream}[\kappa] \). Then, sequences are defined using the clock quantifier, sequence \( \equiv \forall \kappa. \triangleright^\kappa \text{stream}[\kappa] \). Then it is possible to define the anti-causal function that drops every other element:

\[
\begin{align*}
evens : \text{sequence} \rightarrow \text{sequence} & \quad \text{tail} : \text{sequence} \rightarrow \text{sequence} \\
evens \equiv \text{fix}(\lambda F. \lambda \alpha. (\text{fst}(\alpha), F(\text{tail}(\text{tail}(\alpha)))))) & \quad \text{tail} \equiv \lambda \alpha. \text{force}(\text{snd}(\alpha))
\end{align*}
\]

To see that these terms are well-typed, it is necessary to exploit the strong type equality axioms given above.

### 1.3 Guarded Dependent Type Theory

The standard model of guarded recursion without clocks is the topos of trees \( \mathcal{O} \), the presheaves on the poset of natural numbers regarded as a category (Birkedal et al. 2011). This topos can be regarded as a denotational model for a variant of Martin-Löf’s extensional type theory equipped with the \( \triangleright \) 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 GDTRT (Bizjak et al. 2016). In order to justify a crucial clock irrelevance principle, it is necessary to index universes in clock contexts, i.e. \( \mathcal{U}_\Delta \).

In the dependent setting, some difficulties arise when devising a syntax for the semantic type theory of this indexed category. In order to make sense of the “delayed application” operator \( \odot \) in the context of dependent function types, it was necessary to introduce a notion of delayed substitution \( \xi \equiv [\bar{x} \leftarrow \bar{e}] \) which pervades the term language, introducing term formers like \( \triangleright^\kappa \xi.A \) and \( \text{next}^\kappa \xi.e \). 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 GDTT has not yet been proposed at the time this article was written; as such, a canonicity theorem for this system is still forthcoming.

Another challenge is that a computational interpretation of GDTT seems to require the existence of a “fresh clock name generation” effect, in the style of nominal type theory (Pitts 2010; Rahli and Bickford 2016; Schöpp and Stark 2004). On the other hand, the existence of a straightforward operational semantics is also important for concrete implementations of type theory.

Additionally, because GDTT is an extensional type theory, it can enjoy neither a decidable typing relation, nor strong normalization in the presence of an empty type. Therefore, traditional implementation strategies for type theory as deployed in Agda (Norell 2009), Coq (The Coq Development Team 2016), Epigram (McBride 2005) or Idris (Brady 2013) must be revised in favor of techniques whose efficacy do not depend on decidability of judgments, such as those developed in the Nuprl System (Constable et al. 1986) and more recently RedPRL (Sterling et al. 2016) and Andromeda (Bauer et al. 2016). What matters most is not whether types can be checked decidable, but rather the ergonomics of whatever system is employed toward establishing well-typedness of programs.

1.4 Guarded Cubical Type Theory

One way to achieve a decidable typing judgment for GDTT is to adopt an intensional equality, and replace various judgmental principles with propositional axioms (such as the unfolding rule for fix, as well as several other principles having to do with identity types which are validated in extensional GDTT). However, because these axioms destroy the canonicity of the type theory, they should be regarded as a non-starter.

A more refined and well-behaved version of this idea can be found in Guarded Cubical Type Theory (GCTT) by Birkedal et al. (2016), where fix is actually exhibited as a higher-dimensional term, a line or path between a formal fixed point and its one-step unfolding.

GCTT currently supports only a single clock, but it is plausible that it could be extended in the same way as GDTT extends the internal type theory of the topos of trees. Whilst GCTT does not at the time of writing have a decidable typing result, nor a strong normalization theorem, we are confident that these can be achieved in the future in light of the intensional judgmental equality and the restricted unfoldings of fixed points.

1.5 Clocked Type Theory

Recently, an alternative to GDTT called Clocked Type Theory (CloTT) has been proposed, which enjoys a computational interpretation with a canonicity result (Bahr et al. 2017); it is plausible that Clocked Type Theory shall have a decidable typing relation. Notably, Clocked Type Theory does not validate the crucial 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. However, we are interested in the realizability tradition of type theory, in which types are interpreted as behaviors which classify programs rather than as abstract sets which are formed simultaneously with their “elements”. In behavioral type theory, general recursive programs can be written and shown to be (causal, productive, total) in a semantical sense, using the type theory as a program logic; as such, we have prioritized ordinary syntax and ordinary operational semantics in our development.

We perceive that virtue lies in pursuing the intrinsic path, but our practical intentions are oriented toward programming, and the equational theory of derivations in guarded-recursive logic is not our primary object of study.
1.6 Guarded Computational Type Theory

We contribute a new extensional and extrinsic dependent type theory \(\mathsf{CTT}_\ominus\) for guarded recursion and clocks in the Nuprl tradition, enjoying the following characteristics:

1. A straightforward, deterministic operational semantics which induces an immediate canonicity result for base types (Theorem 4.1). This justifies viewing guarded dependent type theory as a programming language, in addition to a theory of mathematical construction.
2. A later modality \(\uparrow_\kappa\ A\) which requires no explicit constructors (next) or destructors (force, \(\otimes\)) in terms.
3. A clock intersection type \(\Lambda_\kappa\ A\) enjoying the clock irrelevance principle, and whose elements require no explicit abstraction \((\Lambda x.e)\) or application \((e[\kappa])\) forms.
4. Guarded fixed points formed as in \(\lambda\)-calculus using the \(Y\)-combinator, having the type \((\uparrow_\kappa\ A \to A) \to A\).
5. A rich and extensional equational theory at the judgmental level, enabling the use of \(\mathsf{CTT}_\ominus\) as a program logic for constructing and reasoning about recursive functions on infinite data.
6. A predicative hierarchy of universes \(U_i\), free of indexing by clock contexts.

Finally, in the spirit of Nuprl, we present an implementable proof refinement logic \(\ominus\mathsf{PRL}\) and program calculus (Section 5) for interactively developing well-typed programs simultaneously with their typing derivations.

2 PROGRAMMING IN GUARDED COMPUTATIONAL TYPE THEORY

Following the computational meaning-theoretic tradition initiated by Martin-Löf (1979), and developed further in the Nuprl project (Allen et al. 2006), we build Guarded Computational Type theory \((\mathsf{CTT}_\ominus)\) on the basis of an untyped programming language of realizers, whose syntax is summarized in Figure 1.

Syntax. The grammar includes operators for both terms and types, which are not distinguished syntactically in any way. Typeness, equality and type membership are semantic properties which will be imposed after we propound the meaning explanation in Section 3. We include syntax for dependent function types \((x : A) \to B\), dependent pair types \((x : A) \times B\), extensional equality types \(\mathsf{Eq}_A(e_1; e_2)\), clock-indexed later modalities \(\uparrow_\kappa\ A\), clock intersection types \(\Lambda_\kappa\ A\), booleans, natural numbers, and a countable hierarchy of type universes \(U_i\). We define the following derived forms for non-dependent function and pair types:

\[
A \to B \triangleq (x : A) \to B \\
A \times B \triangleq (x : A) \times B
\]

Free variables and clocks. We will write \(\mathsf{FreeVars}(e)\) for the set of variables free in the term \(e\), and \(\mathsf{FreeClocks}(e)\) for the set of clock names free in \(e\). A term \(e\) is called closed when \(\mathsf{FreeVars}(e) = \emptyset\); such a term is permitted to have free clock variables. We will write \(\mathsf{ctm}\) for the collection of such closed terms.

Clock substitutions. Terms are subject to general substitutions of clock names for clock names. For clock contexts \(\Lambda, \Lambda'\), we write \(\rho : \Lambda' \to \Lambda\) for a substitution of clock names from \(\Lambda'\) for clock names from \(\Lambda\); such a substitution is generated by a combination of permutations (exchange), diagonals (contraction) and projections (weakening). Given a substitution \(\rho : \Lambda' \to \Lambda\) and a term \(e\) such that \(\mathsf{FreeClocks}(e) \subseteq \Lambda\), we have a term \(\rho^* e\) such that \(\mathsf{FreeClocks}(\rho^* e) \subseteq \Lambda'\).

Forming fixed points and primitive recursors. General fixed points can be programmed in \(\mathsf{GCTT}\) exactly as in the untyped \(\lambda\)-calculus, and these can be used to define a primitive recursor which we will eventually show to realize the induction principle for the natural numbers:

\[
\text{fix}(x. e[x]) \triangleq (\lambda x. e[x])(\lambda x. e[x]) \\
\text{natrecc}(\epsilon_n; e_z; x, y, e_y[x, y]) \triangleq \text{fix}(f. \lambda x. \text{ifze}(x; e_z; y, e_y[y, f(y)]))(\epsilon_n)
\]

As an example, we can now write a program to compute the type of guarded streams for some clock name \(\kappa\) and the type of sequences, using the fixed point combinator which we have just defined in concert with the later
κ ::= κ  (clocks)
e, A ::= x | λx. e | e(e) | (e, e) | fst(e) | snd(e)  (terms)
    ★ | tt | ff | if(e; e; e) | ze | su(e) | ifze(e; e; x, e)
    (x : A) → B | (x : A) × B | EqA(e; e)
    ▶κ A | ▲κ A | bool | nat | U_i
Λ ::= · | Λ, κ  (clock contexts)
Γ, Δ ::= · | Γ, x : A  (typing contexts)

Fig. 1. The syntax of Guarded Computational Type Theory (CTTₒ). Terms e are identified up to renamings of their bound variables.

modality and the clock intersection type:

stream[x] ≜ fix(A. nat × ▶κ A)  
sequence ≜ ▲κ. stream[x]

At the current stage in the definition of CTTₒ, stream[x] and sequence are nothing more than programs, and have no special significance as types. After defining our type system, we will see that these expressions in fact denote types in Section 4.2.

Operational semantics. In Figure 2 we define a small-step operational semantics for the terms of CTTₒ. These definitions specify exactly how untyped programs compute, based on two forms of judgment:

(1) e val which means that the variable-closed term e is a value or canonical form. Crucially, e is allowed to have clock names κ free in it.

(2) e ⇔ e′ means that the variable-closed term e steps (computes) in one step to e′.

Then, using these two forms of judgment as a basis, we write e ⇔* e′ for the reflexive-transitive closure of e ⇔ e′; moreover, we will write e ⇔ e′ for the symmetric closure of e ⇔* e′, i.e. (e ⇔* e′) ∨ (e′ ⇔* e).

We have arranged for the operational semantics to be deterministic, in the sense the stepping judgment can be regarded as a partial endofunction on closed terms; this restriction is essential for justifying the definition of CTTₒ, though it could be lifted if we sufficiently adjusted our meaning explanation (see for instance a remark in Howe (1991) to this end).

As can be seen, the apparatus for guarded recursion and clocks has shown up only in the syntax of types, and does not have any role to play in programming at the computational level. This is because we wish to treat CTTₒ as an extrinsic program logic for reasoning about the behavior of ordinary programs using guarded recursion as a tool; this distinguishes CTTₒ from other type systems such as GDTT, GCTT and CloTT, which treat guarded recursion itself as a fundamental object of study.

Open computation. The operational semantics given in Figure 2 is uniformly extended to open terms in the following way:

\[
\begin{align*}
X ⊢ e ⇔ e′ & \text{ presupposing } \text{FreeVars}(e, e′) ⊆ X \\
X ⊢ e ⇔ e′ & \triangleq ∀γ : \text{ctm}_X. e • γ ⇔ e′ • γ
\end{align*}
\]

All the reduction rules of Figure 2 can be extended uniformly to the open computation judgments; for instance, we can justify rules like the following:

\[
\begin{align*}
\bar{X} \vdash \text{fst}(e) ⇔ \text{fst}(e′) \\
\bar{X} \vdash \text{snd}(e) ⇔ \text{snd}(e′) \\
\bar{X} \vdash \text{fst}((e_1, e_2)) ⇔ e_1 \\
\bar{X} \vdash \text{snd}((e_1, e_2)) ⇔ e_1
\end{align*}
\]
Fig. 2. The operational semantics of \(\text{CTT}_\oplus\).

The form of computational equivalence which we have defined here is equivalence under weak head reduction for open terms; it would also be possible to extend this to a full account of computational congruence using Howe’s Method (Howe 1989) or biorthogonality (Pitts 2004).

3 GUARDED MEANING EXPLANATIONS

The judgments of \(\text{CTT}_\oplus\) should be understood as semantic predicates on the computational behavior of programs, rather than proof-theoretic judgments defined by a recursively enumerable collection of rules. In the spirit of separation logic and other program logics, whatever universally valid patterns of reasoning which we distinguish as rules will necessarily be non-exhaustive and subject to extension as practice demands new inference schemata.

The two fundamental forms of judgment in \(\text{CTT}_\oplus\) are equality of types \(A \cong B\) type, and equality of elements \(e_1 = e_2\), which presupposes \(A \cong A\) type; that is, the membership judgment is only meaningful in case the presupposed typehood judgment is correct. In the spirit of PER semantics, we take equality as the fundamental aspect and treat typehood \(A\) type and membership \(e \in A\) as reflexive cases of equality.

In Martin-Löf (1979), an informal explanation of these judgments was given which was then later explained in mathematical language by Allen (1987) and also by Harper (1992), in which the meaning of these judgments is parameterized over a type system. Allen’s method, developed further by Crary (1998), has been used to formalize the current semantics of Nuprl in Coq (Anand and Rahli 2014); Harper’s method has been used to define a substructural version of CTT for internal imperative programming (Krishnaswami et al. 2015), as well as a generalization of CTT to higher dimensional types (Angiuli et al. 2017).

In our presentation, we will roughly follow Allen’s original definition of type systems, which allows us to avoid making any strong foundational commitments beyond those which are sensible in an arbitrary topos.
3.1 A suitable semantic universe

The semantics of CTT_ω are indexed in a category of clock contexts and clock context morphisms, a technique called Kripke Logical Relations. To avoid the bureaucratic aspects of working with such an encoding and increase the clarity and simplicity of our definitions and proofs, we prefer to work synthetically within a suitable presheaf topos which we will call $S_\Box$.

This move allows us to use ordinary (non-indexed) logical relations naïvely, so long as we work constructively, since relations formed inside a presheaf topos are automatically monotone when viewed from the outside. This idea of relating naïve, internal constructions to the external mathematical reality is codified in the Kripke-Joyal semantics, which (in the special case of sheaf and presheaf toposes) systematically explain what every internal statement means as an external statement about the underlying site or base category (Mac Lane and Moerdijk 1992).

Using this technique, we can no more faithfully realize the design principle which pervades the Martin-Löf tradition, that types should do no more than directly internalize into syntax the structures which are already present in the judgmental base, a basic doctrine which is easily lost to bureaucracy in indexed versions of PER semantics or meaning explanations.

We will require the following things to exist in $S_\Box$:

1. An object $\mathbb{K} : S_\Box$ of clock names.
2. A family of logical modalities $\triangleright_\kappa \phi$ for clock names $\kappa : \mathbb{K}$ and predicates $\phi$ in $S_\Box$.

When we define $S_\Box$, we will arrange for the following principles to hold in its internal logic:

\[
\begin{align*}
\alpha : X & \mid \forall \kappa : \mathbb{K}. \phi(\alpha) \vdash \phi(\alpha) & \text{(Irrelevance, Theorem A.1)} \\
\alpha : X & \mid \forall \kappa : \mathbb{K}. \triangleright_\kappa \phi(\kappa, \alpha) \vdash \forall \kappa : \mathbb{K}. \phi(\kappa, \alpha) & \text{(Forcing, Theorem A.3)} \\
\kappa : \mathbb{K}, \alpha : X & \mid \phi(\kappa, \alpha) \vdash \triangleright_\kappa \phi(\kappa, \alpha) & \text{(Monotonicity, Theorem A.3)} \\
\kappa : \mathbb{K}, \alpha : X & \mid \triangleright_\kappa (\phi(\kappa, \alpha) \Rightarrow \psi(\kappa, \alpha)), \triangleright_\kappa \phi(\kappa, \alpha) \vdash \triangleright_\kappa \psi(\kappa, \alpha) & \text{(Distribution, Theorem A.5)} \\
\kappa : \mathbb{K}, \alpha : X & \mid \triangleright_\kappa \phi(\kappa, \alpha) \Rightarrow \phi(\kappa, \alpha) \vdash \phi(\kappa, \alpha) & \text{(Löb Induction, Theorem A.6)}
\end{align*}
\]

To construct $S_\Box$ as a topos of presheaves, first define $F_\Box : \text{Cat}$ as the free category with strictly associative binary products generated by a single object; explicitly, objects of $F_\Box$ are $U \equiv \bullet^n$ for $n > 0$. A map $f : \bullet^n \rightarrow \bullet^m$ is a projection, but can dually be regarded as an inclusion of finite sets $\mathbb{N}_{<m} \hookrightarrow \mathbb{N}_{<n}$.

Observe that the opposite category $F_\Box^{op}$ is a skeleton of the category of non-empty finite sets and all functions between them. $F_\Box$ is also a full subcategory of $F : \text{Cat}$, 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).

**Remark 3.1.** The category of presheaves $F_\Box^{op}$ is equivalent to the sheaf subcategory of $\mathbb{P}$ under the coverage generated by singleton families of epimorphisms (Staton 2007). This sheaf subcategory is completely analogous to the Schanuel topos (i.e. the category of nominal sets), except that names are subject to identification/contraction. When names are used to represent clocks, this phenomenon has been referred to as “synchronization” (Bizjak and Møgelberg 2015).

Next, define a functor $\otimes[-] : F_\Box \rightarrow \text{Pos}$ (with $\text{Pos}$ the category of partially ordered sets) which will interpret assignments of times to clock names:

\[
\otimes[-] : F_\Box \rightarrow \text{Pos}
\]

\[
\otimes[U] : \omega^{\left|U\right|}
\]

\[
\otimes[f : U \rightarrow \bullet^n](\partial_U : \omega^{\left|U\right|})(\kappa < n) \equiv \partial_U(f(\kappa))
\]

The action of $\otimes[-]$ on objects takes a finite and non-empty cardinality of clock names $U : F_\Box$ 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 (Crole 1993) we can build the total category \( \mathcal{O} : \text{Cat} \triangleq \int^{\mathcal{F}_+} \mathcal{O}[-,-] \) in the following way. Objects are pairs \((U : \mathcal{F}_+, \partial_U : \mathcal{O}[U])\), i.e. collections of clock names together with an assignment; morphisms \( f : (V, \partial_V) \to (U, \partial_U) \) are \( \mathcal{F}_+ \)-morphisms \( f : V \to U \) such that \( \mathcal{O}[f](\partial_V) \leq \partial_U \) in \( \mathcal{O}[U] \). At this time it will be helpful to impose some notation: we will write \( \ell : \mathcal{O} \to \mathcal{F}_+ \) for the induced projection functor, and we will use boldface letters \( U, V \) to range over objects \((U, \partial_U), (V, \partial_V) : \mathcal{O}\).

The semantic universe \( S_{\mathcal{O}} \). Finally, we define our semantic universe as the presheaf topos \( S_{\mathcal{O}} \triangleq \hat{\mathcal{O}} \). This “topos of clocks” defined above inherits a rich internal logic which corresponds to a combination of cartesian/structural nominal indexing\(^1\) and guarded recursion.

The topos \( S_{\mathcal{O}} \) is related to the models considered by Bizjak and Møgelberg (2015), 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.

One further difference is that in order to close the internal logic of \( S_{\mathcal{O}} \) under the clock irrelevance rule described above, we decided to rule out empty clock contexts; this condition, which is equivalent to taking a sheaf subtopos of the presheaves over all clock contexts, was not necessary in the clock-context-indexed presentation of Bizjak and Møgelberg (2015). Our decision to combine clock contexts \( U : \mathcal{F}_+ \) and their time assignments \( \partial_U : \mathcal{O}[U] \) into a single base category \( \mathcal{O} \) has enabled us to avoid indexing universes in clock contexts, which was a necessary condition for \textsc{Gdtt} to have satisfied the clock irrelevance principle.

We have learned that in order to solve the coherence problem for their type theory\(^2\) it will be necessary for the authors of \textsc{Gdtt} to combine the names and time assignments into a single base category, and then work with a full subcategory of the induced presheaf topos over this category in the same way (see Remark 3.1) that we have proposed here.

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

Notations and morphisms. Informally, we will use schematic variables \( \kappa, \kappa', \ldots \) to refer to specific clock names (which formally are indices \( \kappa \equiv i < |U| \) for some \( U : \mathcal{O} \)). As an abuse of notation, we will also use the letter \( \kappa \) to refer to elements of \( \mathbb{K} \), which are at each world \( U \) in bijective correspondence with the cardinality of \( U \). Finally, 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 \( [\kappa \xrightarrow{t} 1] : U \to U[\kappa \mapsto \partial_U(\kappa) + 1] \).

Defining the \( \vDash_{\kappa} \) modalities. We define the \( \vDash_{\kappa} \) modalities by their forcing clause in the Kripke-Joyal semantics of \( S_{\mathcal{O}} \):

\[
\begin{align*}
U \models \vDash_{\kappa} \phi(\alpha) & \triangleq \\
& \begin{cases} 
T & \text{if } \partial_U(\kappa) \equiv 0 \\
U[\kappa \mapsto n] \not\models \phi([\kappa \xrightarrow{1} t] \alpha) & \text{if } \partial_U(\kappa) \equiv n + 1
\end{cases}
\end{align*}
\]

All the other forcing clauses are completely standard; for a reference on Kripke-Joyal forcing, see Mac Lane and Moerdijk (1992).

---

\( ^1 \)That is, the logic of nominal substitution sets (Gabbay and Hofmann 2008; Staton 2007).

\( ^2 \)Personal communication with Bizjak, February 2017.
Meaning explanations vs denotational semantics. The meaning of constructions in computational type theory is quite different from the category-theoretic or denotational view of type theory; rather than being a syntactic presentation of the internal type theory of an arbitrary topos, computational type theory can be regarded as a construction within the subobject lattice of such a topos, i.e. a relational interpretation of a concrete programming language which uses the topos’s internal logic in a definitive capacity. Indeed, implicitly we have used the real internal type theory of the topos $S\Box$ in our definition of the computational type theory $\text{CTT}_\Box$.3

This perspective on computational type theory encompasses within a single framework not only standard $\text{CTT}$ but also forcing extensions of $\text{CTT}$ and $\text{MLTT}$ based on “Beth-Kripke Logical Relations” as presented by Coquand and Mannaa (2016).

3.2 Type systems
At a high level, a type system in the sense of Allen (1987) is an object which distinguishes some terms as types, and specifies what terms will be the elements of those types, and when they will be considered equal. We define a possible type system to be a relation $\mathcal{T} : \mathcal{P}(\text{ctm} \times \mathcal{P}(\text{ctm} \times \text{ctm}))$ in $S\Box$, where ctm is collection of all variable-closed terms. We will write $\text{PTS}$ for the collection of such possible type systems, i.e. $\text{PTS} : S\Box \triangleq \mathcal{P}(\text{ctm} \times \mathcal{P}(\text{ctm} \times \text{ctm}))$.

Let us now define notation for some assertions about possible type systems $\mathcal{T} : \text{PTS}$:

$$\mathcal{T} \models A \sim B \triangleq \exists \mathcal{A} : \mathcal{P}(\text{ctm} \times \text{ctm}). (A, \mathcal{A}) \in \mathcal{T} \land (B, \mathcal{A}) \in \mathcal{T}$$

$$\mathcal{T} \models e_1 \sim e_2 \in A \triangleq \exists \mathcal{A} : \mathcal{P}(\text{ctm} \times \text{ctm}). (A, \mathcal{A}) \in \mathcal{T} \land (e_1, e_2) \in \mathcal{A}$$

A possible type system $\mathcal{T} : \text{PTS}$ is called a type system iff it is the graph of a partial function $\text{ctm} \rightarrow \text{per}(X)$: that is, for all $A : \text{ctm}$ and $\mathcal{A}, \mathcal{B} : \mathcal{P}(\text{ctm} \times \text{ctm})$ such that $(A, \mathcal{A}) \in \mathcal{T} \land (A, \mathcal{B}) \in \mathcal{T}$, we have $\mathcal{A} \equiv \mathcal{B}$ and $\mathcal{A}$ is a partial equivalence relation on $X$. By partial equivalence relation, we mean a relation which is both symmetric and transitive. We write $\text{TS} : S\Box$ for the collection of such type systems.

We can close a binary relation on closed terms under head reduction and head expansion in the following way:

$$\equiv : \mathcal{P}(\text{ctm} \times \text{ctm}) \rightarrow \mathcal{P}(\text{ctm} \times \text{ctm})$$

$$\mathcal{A} \triangleq \{(e_1, e_2) \mid \exists (e'_1, e'_2) \in \mathcal{A}. (e_1 \leftrightarrow e'_1) \land (e_2 \leftrightarrow e'_2)\}$$

This closure operation can be extended to (possible) type systems as well:

$$\equiv : \text{PTS} \rightarrow \text{PTS}$$

$$\mathcal{T} \triangleq \{(A, \mathcal{A}) \mid \exists (A', \mathcal{A}') \in \mathcal{T}. (A \leftrightarrow A') \land (\mathcal{A} \equiv \mathcal{A}')\}$$

Finally, a computational type system is a type system $\mathcal{T} : \text{TS}$ which is closed in the sense that $\mathcal{T} \equiv \mathcal{T}$.

3.3 The judgments of $\text{CTT}_\Box$
Relative to a possible type system $\mathcal{T} : \text{PTS}$, we are now equipped to define the standard judgments of type theory for $\text{CTT}_\Box$.

3Each topos gives rise to two non-equivalent notions of “logic”, namely the logic of the subobject fibration (logic qua logic) and the logic of the codomain fibration (logic qua type theory); this perspective is developed in Jacobs (1999) and Taylor (1999). Following the thought of Lawvere (1971), we reserve the term logical for the former and consider the latter to embody geometrical or mathematical construction in general.

It is also likely possible to regard $\text{CTT}_\Box$ as a fragment of the internal type theory of some realizability topos (van Oosten 2008), but this perspective has not been worked out in detail. MLTT has models in realizability (Beeson 1980; Troelstra and van Dalen, Dirk 1988), but we hesitate to say that it is realizability in exactly that sense.
Categorical judgments. We begin with the categorical judgments, which differ from the $\sim$-judgments defined above only in their presuppositions. The presupposition to a judgment expresses its range of significance: the circumstances under which the judgment has a meaning.\(^4\)

The definitions of the categorical judgments which we give differ from those of Martin-Löf (1979) and Martin-Löf (1984) in that rather than considering types to be generated by relations on values, we interpret them as relations on arbitrary programs which respect head reduction and head expansion. This relaxation of the meaning explanation will enable us to account for the \(\triangleright\) types, which will include elements which are only later evaluable.

1. To know \(T \models A \triangleq B\) type (pronounced "\(A\) and \(B\) are equal types") is to know \(T \models A \sim B\). We will write \(T \models A\) type for \(T \models A \triangleq A\) type.

2. To know \(T \models e_1 \triangleq e_2 \in A\) (pronounced "\(e_1\) and \(e_2\) are equal members of \(A\)") presupposing \(T \models A\) type, is to know \(T \models e_1 \sim e_2 \in A\). We will write \(T \models e \in A\) for \(T \models e \triangleq e \in A\).

Hypothetico-general judgment. We now define a form of hypothetical judgment which expresses semantic consequence with sequential functionality. We write these judgments using the notation \(\Gamma \gg J\) rather than \(\Gamma \vdash J\) in order to avoid confusion with the more common formal/proof theoretic consequence, which is syntactic in nature. These forms of judgment are defined by mutual induction.

1. \(T \models \Gamma \triangleright \text{ctx}\) (pronounced "\(\Gamma\) is a valid context") presupposes nothing.

2. \(T \models \gamma_1 \triangleq \gamma_2 \in^* \Gamma\) (pronounced "\(\gamma_1\) and \(\gamma_2\) are equal instantiations for context \(\Gamma\)") presupposes \(T \models \Gamma \triangleright \text{ctx}\).

3. \(T \models \Gamma \gg A \triangleq B\) type (pronounced "\(A\) and \(B\) are equal types in context \(\Gamma\)") presupposes \(T \models \Gamma \triangleright \text{ctx}\) and \(\text{FreeVars}(A, B) \subseteq [\Gamma]\).

4. \(T \models \Gamma \gg e_1 \triangleq e_2 \in A\) (pronounced "\(e_1\) and \(e_2\) are equal members of type \(A\) in context \(\Gamma\)"") presupposes \(T \models \Gamma \gg A\) type and \(\text{FreeVars}(A, B) \subseteq [\Gamma]\).

Next, we specify the verification conditions for the instances of each of the above judgment forms.

\[
\begin{array}{c}
T \models \cdot \triangleright \text{ctx} & \quad T \models \Gamma \triangleright \text{ctx} \quad T \models \Gamma \gg A \text{ type} \\
T \models [] \triangleq [] \in^* \cdot \\
\end{array}
\]

\[
\begin{array}{c}
T \models y_1 \triangleq y_2 \in^* \Gamma & \quad T \models e_1 \triangleq e_2 \in A \cdot y_1 \\
T \models [y_1, x \leftrightarrow e_1] \triangleq [y_2, x \leftrightarrow e_2] \in^* \Gamma, x : A \\
\forall y_1, y_2 : \text{ctm}[\Gamma], (T \models y_1 \triangleq y_2 \in^* \Gamma) \Rightarrow (T \models A :: y_1 \triangleq B :: y_2 \text{ type}) \\
T \models \Gamma \gg A \triangleq B \text{ type} \\
\forall y_1, y_2 : \text{ctm}[\Gamma], (T \models y_1 \triangleq y_2 \in^* \Gamma) \Rightarrow (T \models e_1 :: y_1 \triangleq e_2 :: y_2 \in A :: y_1) \\
T \models \Gamma \gg e_1 \triangleq e_2 \in A
\end{array}
\]

Finally, for convenience we define a new form of judgment which quantifies over clocks. For any categorical judgment \(J\) and clock context \(\Lambda\), define:

\[T \models \Lambda \gg J \triangleq \forall \Lambda \cdot \cdots : \Xi. T \models \Gamma \gg J\]

\(^4\)To be more precise, in Martin-Löf’s meaning-theoretic method, a form of judgment \(J\) is laid down first specifying the conditions under which an instance of \(J\) has a meaning—and these conditions are called its presupposition; then, for each instance of \(J\) for which this presupposition holds, the conditions under which this instance of \(J\) are held to be evident are specified. Together, these definitions constitute a meaning explanation. The concept of presupposition in the Martin-Löf tradition is also explained in more formal terms by Schroeder-Heister (1987).
For the following, fix a computational type system $T : TS$.

**Theorem 3.2 (Closed computation).** We have the following head expansion and head reduction results for categorical judgments:

1. If $A \leftrightarrow A'$ and $T \models A \equiv B$ type, then $T \models A' \equiv B$ type.
2. If $e_1 \leftrightarrow e_1'$ and $T \models e_1 = e_2 \in A$, then $T \models e_1' = e_2 \in A$.

**Proof.** (1) There exists some $\mathcal{A} : \text{per(ctm)}$ such that $(A, \mathcal{A}) \in T$ and $(B, \mathcal{A}) \in T$. We need to show that there exists some $\mathcal{A}' : \text{per(ctm)}$ such that $(A', \mathcal{A}') \in T$ and $(B, \mathcal{A}') \in T$. We will choose $\mathcal{A}' = \mathcal{A}$, which is the same as to say $\mathcal{A}' \equiv \mathcal{A}$ because $T$ is computational. It remains only to show that $(A', \mathcal{A}) \in T$; because $T$ is computational, it suffices to show that $A' \leftrightarrow A$ and $(A, \mathcal{A}) \in T$, both of which we have assumed.

(2) There exists some $\mathcal{A} : \text{per(ctm)}$ such that $(A, \mathcal{A}) \in T$ and $(e_1, e_2) \in \mathcal{A}$. We need to show that $(e_1', e_2) \in \mathcal{A}$. Because $T$ is computational, $\mathcal{A} \equiv \mathcal{A}$; therefore, because $e_1 \leftrightarrow e_1'$, we have $(e_1', e_2) \in \mathcal{A}$. □

**Theorem 3.3.** Respect for computation extends to hypothetico-general judgments over open terms as well:

1. If $[\Gamma] \vdash A \leftrightarrow A'$ and $T \models \Gamma \gg A \equiv B$ type, then $T \models \Gamma \gg A' \equiv B$ type.
2. If $[\Gamma] \vdash e_1 \leftrightarrow e_1'$ and $T \models \Gamma \gg e_1 = e_2 \in A$, then $T \models \Gamma \gg e_1' = e_2 \in A$.

**Proof.** (1) Fixing $T \models y_1 \equiv y_2 \in^* \Gamma$, we need to show that $T \models A' \cdot y_1 \equiv B \cdot y_2$ type under the assumption that $T \models A \cdot y_1 \equiv B \cdot y_2$ type. From $[\Gamma] \vdash A \leftrightarrow A'$ we have $A \cdot y_1 \leftrightarrow A' \cdot y_2$, from which by Theorem 3.2 we have our goal.

(2) Fixing $T \models y_1 \equiv y_2 \in^* \Gamma$, we need to show that $T \models e_1 \cdot y_1 \equiv e_2 \cdot y_1 \in A \cdot y_1$, under the assumption that $T \models e_1 \cdot y_1 \equiv e_2 \cdot y_1 \equiv e_2 \cdot y_1 \in A \cdot y_1$. From $[\Gamma] \vdash e_1 \leftrightarrow e_1'$, we have $e_1 \cdot y_1 \leftrightarrow e_1' \cdot y_1$, and so our goal follows from Theorem 3.2. □

### 4 THE DEFINITION OF TYPES

In the previous sections, we have defined the notion of type system and given semantical meaning explanations for all the judgments of $\text{CTT}_0$ relative to a possible type system $T : \text{PTS}$. Next, we will show how to close a possible type system under the type formers of $\text{CTT}_0$, namely booleans, natural numbers, dependent functions types, dependent pair types, equality types, later modalities and clock intersection types. We follow Anand and Rahli (2014) and Crary (1998) in presenting the closure operator $c[-] : \text{PTS} \rightarrow \text{PTS}$ below as an inductive definition.
Remarks on $c[-]$. A few remarks are in order, especially in regard to the later modality types $\triangleright_k A$. Observe that we have not required that $A$ be a type in order for $\triangleright_k A$ to be a type: we only require that this premise obtain later. In order to ensure that the definition of $A$ in this case is well-formed, we then use the presupposition-free $\sim$-judgment rather than $\triangleright_k$. Observe that if we had written $\triangleright_k(T \models e_1 \equiv e_2 \in A)$, we would have required an additional premise that $T \models A$ type, where we only wanted $\triangleright_k(T \models e_1 \equiv e_2 \in A)$.

Similarly put, this is because $T \models e_1 \equiv e_2 \in A$ is only a well-formed judgment if its presupposition (that $A$ is a type) is satisfied already, whereas $T \models e_1 \sim e_2 \in A$ requires no such condition to hold a priori: at whatever stage $A$ fails to be a type, $T \models e_1 \sim e_2 \in A$ will be a non-evident but still well-formed statement.

Constructing the type system hierarchy. In terms of our closure operator, we can now define a sequence of type systems which captures every prefix of the $\mathbb{CTT}_\infty$'s universe hierarchy:

$$
\begin{align*}
U[-] : \mathbb{N} & \to \mathbb{TS} \\
U & \equiv \{(A, A') \mid c[U][i] \models A \equiv A' \text{ type}\} \\
& \quad \text{(} U_i, U \text{)} \in \mathbb{U}[i + 1] \\
(A, A) & \in U[i]
\end{align*}
$$

By taking the join of this sequence of type systems, we can construct the type system which contains the full universe hierarchy, $U_{\omega} : \mathbb{TS}$. Then by closure, we construct the ultimate type system $T_{\omega} : \mathbb{TS}$ of $\mathbb{CTT}_\infty$.

$$
\begin{align*}
U_{\omega} & : \mathbb{TS} \\
U_{\omega} & \equiv \{(A, A) \mid \exists i : \mathbb{N} \text{. } (A, A) \in U[i]\} \\
T_{\omega} & : \mathbb{TS} \\
T_{\omega} & \equiv c[U_{\omega}]
\end{align*}
$$

Observe that for all $i : \mathbb{N}$, the type system $U[i]$ is a computational type system, from which we can conclude that $U_{\omega}$ is a computational type system, and that $T_{\omega}$ is also a computational type system.
Theorem 4.1 (Canonicity). For any term e, if $T_{\omega} \vdash \Lambda \mid e \in \text{bool}$ then either $e \mapsto \text{tt}$ or $e \mapsto \text{ff}$.

Proof. By definition, $(\text{bool}, \mathcal{A}) \in T_{\omega}$ where $\mathcal{A} \equiv \{(\text{tt}, \text{tt}), (\text{ff}, \text{ff})\}$; because $T_{\omega}$ is a type system, this is also the unique such $\mathcal{A}$. Therefore, unfolding the meaning of the membership judgment and using uniqueness of type relations, we have $(e, e) \in \mathcal{A}$. Unfolding the meaning of the closure of a relation under head expansion and reduction, this means that there exist $(e'_1, e'_2) \in \{(\text{tt}, \text{tt}), (\text{ff}, \text{ff})\}$ such that $e \leftrightarrow e'_1$ and $e \leftrightarrow e'_2$. From this, it is obvious that either $e \mapsto \text{tt}$ or $e \mapsto \text{ff}$. □

4.1 Rules of Inference

On the basis of the computational type system $T_{\omega} : \text{TS}$, it is now possible to interpret a collection of inference rules which are useful in practice for reasoning about guarded recursive programs. These include analogues to all the rules of standard extensional type theory, extended with direct/untyped computation as well as the rules and equational principles which pertain to guarded recursion and clocks. In the remainder of this paper, we will write simply $\mathcal{F}$ for $T_{\omega} \models \mathcal{F}$.

Caution. We consider it important to view these rules as theorems of $\text{CTT}$ (schemata for justified reasoning from premise to conclusion on the basis of our computational semantics) rather than as an inductive definition of a formal system. When we prove the correctness of a rule scheme, we mean to say that the conclusion is evident when the premise is evident; these justifications may elsewhere be deployed in the soundness theorem for a separate formal system, which will naturally proceed by induction. An example of such a formal system is given in Section 5.

In Figure 3, we present the "structural" rules of $\text{CTT}$, which are valid for any type system; these include the genuine structural rules (hypothesis, weakening, exchange, closure under clock substitutions) as well as symmetry and transitivity of equality, and the respect of member equality for type equality. In Figure 4 we present a collection of rules for forming types relative to $T_{\omega}$. Whilst at a semantic level, the principle judgment of interest for types is $A \equiv B \text{ type}$, due to the presence of universe levels, it is less duplicative in practice to specify the type formation rules as universe membership judgments $A \equiv B \in U_i$ and then supply a reflection rule which recognizes every member of a universe as a type.

In Figure 5 we present the declarative member equality rules for the types of $T_{\omega}$; these include both the equality of introduction forms and the structural equality of elimination forms. Most of the rules presented are completely standard in Martin-Löf type theories and require no special justification. Below, we will justify the correctness of some of the rules which distinguish $\text{CTT}$ from other type theories; the remainder of the distinguishing rules are justified in Appendix B.

4.1.1 Löb induction. Recall that $\text{fix}(x. e)$ is simply notation for an instance of the $Y$-combinator. We have asserted the following remarkable rule for typing guarded fixed points relative to $T_{\omega}$:

$$
\Lambda, x : \mathcal{K} \vdash \Gamma \mid e_1 \equiv e_2 \equiv A
$$

We can justify this rule using the Löb Induction principle in our metalogic $\mathcal{S}_\oplus$.

Proof. Fixing concrete clocks $\Lambda \ldots \kappa : \mathcal{K}$ and $\gamma_1 \equiv \gamma_2 \in \Gamma$, we have to show $\text{fix}(x. e_1 \cdot \gamma_1) \equiv \text{fix}(x. e_2 \cdot \gamma_1) \equiv A \cdot \gamma_1$; by a simple calculation, we can push the substitutions underneath the fixed points:

$$
\text{fix}(x. e_1 \cdot \gamma_1) \equiv \text{fix}(x. e_2 \cdot \gamma_2) \in A \cdot \gamma_1 \quad \text{(Goal)}
$$

By Löb Induction in $\mathcal{S}_\oplus$, we may freely assume the delayed version of our goal with respect to our chosen clock $\kappa$:

$$
\mathcal{K}(\text{fix}(x. e_1 \cdot \gamma_1)) \equiv \text{fix}(x. e_2 \cdot \gamma_2) \in A \cdot \gamma_1)
$$
Next, unfold the meaning of our premise: for all \([y_1, x \mapsto f_1] = [y_2, x \mapsto f_2] \in \mathcal{F}, x : \top T\), we have \(e_1 \cdot [y, x \mapsto f_1] \triangleq e_2 \cdot [y, x \mapsto f_2] \in A \cdot \gamma_1\). Because \(x \notin \text{FreeVars}(A)\), this is the same as to say \(e_1 \cdot [y, x \mapsto f_1] = e_2 \cdot [y, x \mapsto f_2] \in A \cdot \gamma_1\). We have to choose \(f_i\) such that \(f_1 \triangleq f_2 \in \top T, \gamma_1\). If we instantiate \(f_i \triangleq \text{fix}(x, e_i \cdot \gamma_1)\), our goal is identical to (Hyp).\(\square\)
4.1.2 Delayed dependent functions. We also have asserted the following subtyping rule for dependent function types:

\[ \Lambda, \kappa \mid \Gamma \gg e_1 \equiv e_2 \in [\kappa \Rightarrow A \Rightarrow B] \]

\[ \Lambda, \kappa \mid \Gamma \gg e_1 \equiv e_2 \in (x : A) \Rightarrow B \]

This rule, which expresses the fact that a delayed function may be applied to a delayed input to return a delayed output, is quite strong in comparison to previous attempts to integrate guarded recursion with dependent types: in particular, CTT requires no special notion of delayed substitution in our syntax in order to formulate such a principle (unlike GDTT).

The reason that a rule like this was not possible in GDTT has to do with the fact that GDTT is a syntactic presentation of (the internal type theory of a certain topos \( \mathcal{T} \)), and the later modalities were regarded as fibered endofunctors on the slices of the topos. Therefore, for \( \triangleright\kappa A \) to be a type (i.e. object of the slice topos), \( A \) had to be a type as well. The problem was that the expression \( (x : \triangleright\kappa A) \rightarrow \triangleright\kappa B \) might not even denote a well-formed type, since \( B \) was defined as a dependent type over \( A \), not over \( \triangleright\kappa A \).

Because of the relational character of CTT’s meaning explanation, we are perfectly free to define our later modality such that \( \triangleright\kappa A \) type even when we have only \( \triangleright\kappa(A \text{ type}) \), i.e. \( A \) shall be a type in one moment. Indeed, from \( \Gamma, x : A \gg B \text{ type} \) we can conclude \( \Gamma \gg (x : \triangleright\kappa A) \rightarrow \triangleright\kappa B \text{ type} \).

Proof. To show that \( \Gamma \gg (x : \triangleright\kappa A) \rightarrow \triangleright\kappa B \text{ type} \), it suffices to show that \( \Gamma, x : \triangleright\kappa A \gg \triangleright\kappa B \text{ type} \). Fixing \( y_1 \equiv y_2 \in \mathcal{T} \) and \( a_1 \equiv a_2 \in \triangleright\kappa A \cdot y_1 \), it suffices to show that \( \triangleright\kappa B[y_1, x \leftarrow a_1] \equiv \triangleright\kappa B[y_2, x \leftarrow a_2] \text{ type} \).

By definition, it remains to show that \( \triangleright\kappa(T_\omega \models B[y_1, x \leftarrow a_1] \sim B[y_2, x \leftarrow a_2]) \), which follows from our original premise \( \Gamma, x : A \gg B \text{ type} \) and the fact that \( \triangleright\kappa(T_\omega \models a_1 \sim a_2) \).
Having established that the presuppositions of the proposed rule are satisfied, we can now proceed to prove its correctness:
\[
\begin{align*}
\Lambda, \kappa \mid \Gamma &\Rightarrow e_1 \equiv e_2 \in \Gamma, (x : A) \rightarrow B \quad \Lambda, \kappa \mid \Gamma, x : A \Rightarrow B \text{ type} \\
\Lambda, \kappa \mid \Gamma &\Rightarrow e_1 \equiv e_2 \in (x : \Gamma, A) \rightarrow \Gamma, B
\end{align*}
\]

**Proof.** Fix clocks $\Lambda \ldots \kappa : \kappa$ and instantiations $y_1 \equiv y_2 \in \Gamma$. Now, we need to show that $e_1 \cdot y_1 \equiv e_2 \cdot y_2 \in (x : \Gamma, A \cdot y_1) \rightarrow \Gamma, B \cdot y_1$, which is by definition the same as to say $x : \Gamma, A \cdot y_1 \Rightarrow e_1(x) \cdot y_1 \equiv e_2(x) \cdot y_1 \in \Gamma, B \cdot y_1$.

Fixing $a_1 \equiv a_2 \in \Gamma, A \cdot y_1$, we need to show that $e_1 \cdot y_1(a_1) \equiv e_2 \cdot y_2(a_2) \in \Gamma, B \cdot [y_1, x \leftarrow a_1]$. By definition, this is the same as to assert that $e_1 \cdot y_1(a_1) \equiv e_2 \cdot y_2(a_2) \in B \cdot [y_1, x \leftarrow a_1]$. By unfolding and instantiating our premise, we have $e_1 \cdot y_1 \equiv e_2 \cdot y_2 \in \Gamma, A \cdot y_1 \rightarrow B \cdot y_1$, which is the same as the following:
\[
\begin{align*}
\begin{array}{c}
\Rightarrow e_1 \cdot y_1 \equiv e_2 \cdot y_2 \in (x : A \cdot y_1) \rightarrow B \cdot y_1
\end{array}
\end{align*}
\]

The above unfolds to a delayed hypothetical judgment:
\[
\begin{align*}
\Rightarrow e_1 \cdot y_1 \equiv e_2 \cdot y_2(x) \in B \cdot y_1 \tag{H}
\end{align*}
\]

From $a_1 \equiv a_2 \in \Gamma, A \cdot y_1$ we have $e_1(a_1) \equiv e_2(a_2) \in A \cdot y_1$; using (H), we have our goal:
\[
\begin{align*}
\Rightarrow e_1(a_1) \equiv e_2(a_2) \in B \cdot [y_1, x \leftarrow a_1]
\end{align*}
\]

\[\square\]

### 4.1.3 Untyped computation

In CTT, we eschew the traditional typed “computation rules” in favor of untyped direct computation rules; this reasoning principle, justified by Theorem 3.3, is codified by the following two rules:

\[
\begin{align*}
\begin{array}{c}
| \Gamma | A \leftrightarrow A' \quad \Lambda | \Gamma \Rightarrow A \equiv B \text{ type} \\
\Lambda | \Gamma \Rightarrow A' \equiv B \text{ type}
\end{array}
\end{align*}
\]

One benefit of these rules is that they are far simpler to use in practice than the typed $\beta$-rules which are more common in presentations of Martin-Löf type theories. Direct computation rules of this kind are valid for the standard semantics of Martin-Löf type theory as presented in Martin-Löf (1979) and Martin-Löf (1984), but are rarely included in formal systems or implementations outside the Nuprl tradition.

It is worth remarking that in semantic and computational type theories, typing obligations do not come for free: they must be proved, since there is no method to decide them in general. As a result, engineers of computational type theories prefer rules which give rise to a minimal number of typing premises.

### 4.1.4 Type extensionality rules for clocks

A crucial aspect of CTT is the validation of a number of very strong type equations up to judgmental equality. These rules, summarized below, reflect into our type theory the analogous principles for the logical operators $\Rightarrow$ and $\forall \kappa : \kappa$ from the topos $S_{\kappa}$.

\[
\begin{align*}
\begin{array}{c}
\text{Irrelevance} \quad \kappa \notin \text{FreeClocks}(A) \\
\text{Forcing} \quad \Lambda, \kappa \mid \Gamma \Rightarrow A \text{ type}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{Irrelevance} \quad \kappa \notin \text{FreeClocks}(A) \\
\text{Forcing} \quad \Lambda, \kappa \mid \Gamma \Rightarrow A \text{ type}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{\kappa Distribution} \\
\text{\kappa Distribution}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{Forcing} \quad \Lambda, \kappa \mid \Gamma \Rightarrow A \text{ type} \\
\text{Forcing} \quad \Lambda, \kappa \mid \Gamma, x : A \Rightarrow B \text{ type}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{\kappa Distribution} \\
\text{\kappa Distribution}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{Forcing} \quad \Lambda, \kappa \mid \Gamma \Rightarrow A \text{ type} \\
\text{Forcing} \quad \Lambda, \kappa \mid \Gamma, x : A \Rightarrow B \text{ type}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{\kappa Distribution} \\
\text{\kappa Distribution}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{Forcing} \quad \Lambda, \kappa \mid \Gamma \Rightarrow A \text{ type} \\
\text{Forcing} \quad \Lambda, \kappa \mid \Gamma, x : A \Rightarrow B \text{ type}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{\kappa Distribution} \\
\text{\kappa Distribution}
\end{array}
\end{align*}
\]
In \textit{GDTT}, similar principles are available but they are formulated as type isomorphisms rather than equations. A consequence of justifying these equations for \textit{CTT}\textsuperscript{\textbullet} is that we do not have to interleave coercions into our programs.

4.2 Revisiting streams and sequences

With the definition of \textit{CTT}\textsuperscript{\textbullet} in hand, we are prepared to revisit our example of guarded streams and coinductive sequences, and justify our constructions with respect to the type system. Recall from Section 2 how we defined guarded streams and sequences:

\[
\text{stream}[\kappa] \triangleq \text{fix}(A. \text{nat} \times \uparrow_{\kappa} A) \quad \text{sequence} \triangleq \bigcap \kappa \. \text{stream}[\kappa]
\]

We can now use the collected rules of \textit{CTT}\textsuperscript{\textbullet} to show that these expressions indeed denote genuine types.

\[
\begin{array}{c}
\frac{\kappa \mid A : \uparrow_{\kappa} U_0, x : \text{nat} \Rightarrow A \in \uparrow_{\kappa} U_0}{\kappa \mid A : \uparrow_{\kappa} U_0 \Rightarrow \text{nat} \in U_0} \\
\frac{\kappa \mid A : \uparrow_{\kappa} U_0, x : \text{nat} \Rightarrow \uparrow_{\kappa} A \in U_0}{\kappa \mid \cdot \Rightarrow \text{stream}[\kappa] \in U_0}
\end{array}
\]

We can also show that sequence \(\triangleq \text{nat} \times \text{sequence} \in U_0\), which will justify the typing of operators to project the head and the tail of a sequence. Rather than explicitly construct the derivation, we will reason informally using the rules of \textit{CTT}\textsuperscript{\textbullet}:

1. By untyped computation, we have \(\kappa \mid \cdot \Rightarrow \text{stream}[\kappa] \triangleq \text{nat} \times \uparrow_{\kappa} \text{stream}[\kappa] \in U_0\).
2. By intersection formation, we have \(\bigcap \kappa \. \text{stream}[\kappa] \triangleq \bigcap \kappa \. (\text{nat} \times \uparrow_{\kappa} \text{stream}[\kappa]) \in U_0\).
3. By intersection distribution, we have \(\bigcap \kappa \. (\text{nat} \times \uparrow_{\kappa} \text{stream}[\kappa]) \triangleq \bigcap \kappa \. \text{nat} \times \bigcap \kappa \. \uparrow_{\kappa} \text{stream}[\kappa] \in U_0\).
4. By clock irrelevance, we have \(\bigcap \kappa \. \text{nat} \triangleq \text{nat} \in U_0\); by the forcing rule, we have \(\bigcap \kappa \. \uparrow_{\kappa} \text{stream}[\kappa] \triangleq \text{sequence} \in U_0\).
5. By pair type formation, we have \(\bigcap \kappa \. \text{nat} \times \bigcap \kappa \. \uparrow_{\kappa} \text{stream}[\kappa] \triangleq \text{nat} \times \text{sequence} \in U_0\).
6. By transitivity of equality and all the above, we have sequence \(\triangleq \text{nat} \times \text{sequence} \in U_0\).

\square

Defining \texttt{zipWith} for streams. As another example, the naïve general recursive implementation of \texttt{zipWith} can be shown to be well-typed as a function on streams in our setting:

\[
\kappa \mid \cdot \Rightarrow \text{zipWith} \in (\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \rightarrow \text{stream}[\kappa] \rightarrow \text{stream}[\kappa] \rightarrow \text{stream}[\kappa]
\]

\[
\text{zipWith} \triangleq \lambda f . \text{fix}(g, \lambda \alpha. \lambda \beta. (f(fst(\alpha))(fst(\beta)), g(snd(\alpha))(snd(\beta))))
\]

1. By function introduction, we have \(f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}\) and need to exhibit an element of \(\text{stream}[\kappa] \rightarrow \text{stream}[\kappa] \rightarrow \text{stream}[\kappa]\).
2. By the Löb induction rule, we may freely assume \(g : \uparrow_{\kappa} \text{stream}[\kappa] \rightarrow \text{stream}[\kappa] \rightarrow \text{stream}[\kappa]\).
3. By function introduction, we assume \(\alpha, \beta : \text{stream}[\kappa]\) and need to exhibit an element of \(\text{stream}[\kappa]\).
4. By direct computation, it suffices to exhibit an element of \(\text{nat} \times \uparrow_{\kappa} \text{stream}[\kappa]\); in the same way, we also have assumptions \(\alpha, \beta : \text{nat} \times \uparrow_{\kappa} \text{stream}[\kappa]\).

   a. For the first projection, we easily have \(f(fst(\alpha))(fst(\beta)) \in \text{nat}\).
   b. For the second projection, we need to exhibit an element of \(\uparrow_{\kappa} \text{stream}[\kappa]\). From the subtyping rule for delayed functions types in Section 4.1.2, we have:

\[
g \in \uparrow_{\kappa} \text{stream}[\kappa] \rightarrow \uparrow_{\kappa} (\text{stream}[\kappa] \rightarrow \text{stream}[\kappa])
\]
Therefore, we have \( g(\text{snd}(\alpha)) \in \triangleright \kappa \) (\text{stream}[\kappa] \rightarrow \text{stream}[\kappa])\), and by applying the subtyping rule again, we have:

\[
g(\text{snd}(\alpha)) \in \triangleright \kappa \text{ stream}[\kappa] \rightarrow \triangleright \kappa \text{ stream}[\kappa]
\]

As a result, we establish our goal using \( g(\text{snd}(\alpha))(\text{snd}(\beta)) \in \triangleright \kappa \) \text{ stream}[\kappa]. \]

\[\square\]

5 PROOF REFINEMENT LOGIC

In this section, we present the prototype of a proof refinement logic in the sense of Bates and Constable (Bates and Constable 1985; Constable et al. 1986; Harper 1985), adapted to handle dependent subgoals using ideas from Sterling and Harper (2017) and Spiwack (2011).

Traditionally, proof assistants based on Martin-Löf type theories are based on a combination of typechecking and elaboration. Since our type theory does not have a decidable judgmental equality or typing relation, we must pursue a different approach. We will build a Proof Refinement Logic (PRL) in the style of Nuprl (Constable et al. 1986), using techniques developed in the RedPRL project (Sterling et al. 2016).

5.1 Proof Refinement Logic

A proof refinement logic is a sequent calculus based fundamentally around a notion of program extraction, equipped with a collection of rules which each specify a decomposition of a main goal (conclusion) into a list of subgoals (premises). Moreover, each rule specifies a construction of a witness for its conclusion relative to constructions of witnesses for its premises. Unlike in Nuprl, later subgoals may depend on the witnesses assumed in previous subgoals.

Algorithmic refinement judgments. A refinement judgment is a form of judgment which has several “input terms” and several “output terms”, which are distinguished in this presentation by color; historically, in the Nuprl tradition, only a single output has been permitted, which is called the extract of the judgment. We write \( \mathcal{J}(\vec{e}) \sim \vec{o} \) to range over an instance of an arbitrary refinement judgment with inputs \( \vec{e} \) and outputs \( \vec{o} \), however we will use more specific notations for the actual refinement judgments of \( \odot \text{PRL} \).

These forms of judgments should be thought of specifying different kinds of problems or tasks; as such, the outputs \( \vec{o} \) is constructed algorithmically and incrementally over the course of these tasks’ fulfillment. In other words, it is the result of asserting the judgment \( \mathcal{J}(\vec{e}) \), as opposed to being part of the assertion.

The refinement judgments of \( \odot \text{PRL} \) largely resemble sequent-calculus versions of the semantic judgments of \( \text{CTT\textsubscript{P}} \) which are oriented toward the implicit construction of witnesses/extracts. In \( \odot \text{PRL} \), we specify the following forms of algorithmic refinement judgment:

1. \( \Lambda \mid \Gamma \Rightarrow \text{type} \ni A_1 \equiv A_2 \; @ i \) signifies the task of constructing two equal types \( A_1, A_2 \) at universe level \( i \), relative to clocks \( \Lambda \) and assumptions \( \Gamma \). The semantics of this judgment are that assuming \( \Lambda \mid \Gamma \text{ctx} \), then we have \( \Lambda \mid \Gamma \Rightarrow A_1 \equiv A_2 \in U_i \).
2. \( \Lambda \mid \Gamma \Rightarrow A \ni e_1 \equiv e_2 \) signifies the task of constructing equal elements \( e_1, e_2 \) of type \( A \) relative to clocks \( \Lambda \) and assumptions \( \Gamma \). The semantics of this judgment are that assuming \( \Lambda \mid \Gamma \text{ctx} \), then we have \( \Lambda \mid \Gamma \Rightarrow e_1 \equiv e_2 \in A \).
3. \( \Lambda \mid \Gamma \Rightarrow A \equiv B \; \text{type} \) signifies the task of establishing that \( A \) and \( B \) are equal types relative to clocks \( \Lambda \) and assumptions \( \Gamma \). Note that this judgment has no extract.\(^5\) The semantics of this judgment are that assuming \( \Lambda \mid \Gamma \text{ctx} \), then we have \( \Lambda \mid \Gamma \Rightarrow A \equiv B \; \text{type} \).

\(^5\)When we view refinement judgments systematically, it is better to say that this judgment has a nullary extract.
(4) $\Lambda \mid \Gamma \Rightarrow e_1 \equiv e_2 \in A$ signifies the task of establishing that $e_1$ and $e_2$ are equal elements of type $A$ relative to clocks $\Lambda$ and assumptions $\Gamma$. The semantics of this judgment are that assuming that $\Lambda \mid \Gamma \text{ctx}$, we have $\Lambda \mid \Gamma \Rightarrow e_1 \equiv e_2 \in A$.

(5) $X \Rightarrow e \mapsto* e'$ signifies the task of reducing the open term $e$ to some term $e'$ relative to free variables $X$. The semantics of this judgment are that $X \vdash e \mapsto* e'$.

Finally, we additionally have some auxiliary forms of refinement judgment which express side conditions, having no outputs/extract:

1. $e_1 \equiv, e_2$ signifies the condition that $e_1$ and $e_2$ are identical.
2. $i \leq j$ signifies the condition that universe level $i$ is less than or equal to universe level $j$.
3. $\kappa \notin \text{FreeClocks}(e)$ signifies the condition that the clock $\kappa$ is not free in the term $e$. Moreover, this condition shall only obtain when $e$ has no free metavariables, i.e. $\text{FreeMetas}(e) \equiv \emptyset$.

Refinement rules. A dependent refinement rule is a schema for decomposing goals into subgoals, implementing program extraction. A collection of such rules comprises a proof refinement logic. For an arbitrary form of refinement judgment $J(\bar{e}) \sim \bar{\delta}$, a refinement rule is given in the following form:

\[
\begin{align*}
J(\bar{e}) & \sim \bar{\delta} \\
\text{by name-of-rule} \\
J_1(\bar{e}_1) & \sim \bar{x}_1 \\
J_2(\bar{e}_2) & \sim \bar{x}_2 \\
& \vdots \\
J_n(\bar{e}_n) & \sim \bar{x}_n
\end{align*}
\]

Metavariables and binding structure. We enrich our term language with a notion of metavariable (schematic variable) $x$, which we write by convention in the German face. For a term $e$, we write $\text{FreeMetas}(e)$ for the set of free metavariables in $e$. In the syntax of refinement rules above, the output of each subgoal $J_i$ binds a vector of metavariables $\bar{x}_i$ which may be used in the outputs $\bar{\delta}$ of the main goal as well as the inputs $\bar{e}_j$ of all subgoals $J_{j>i}$.

Derivability. Derivability with respect to dependent refinement rules is essentially the same as the usual notion of derivability in formal systems, but we will spell it out formally. We write $\text{rule}; [\text{rule}_0, \ldots, \text{rule}_n]$ for the derived rule which applies the rule $\text{rule}_i$ to each $i$th subgoal induced by the rule $\text{rule}_i$.

\[
\begin{align*}
J(\bar{e}) & \sim \bar{\delta} \\
\text{by rule} \\
J_i(\bar{e}_i) & \sim \bar{x}_i \\
\text{by rule}_i \\
J_{ij}(\bar{e}_{ij}) & \sim \bar{x}_{ij} \\
\end{align*}
\]

Multicut
We also include a (suggestively named) primitive identity rule \( \text{hole} \) which reproduces the current goal as follows:

\[
\begin{align*}
\mathcal{I}(\bar{e}) \leadsto x \\
\text{by \ hole?} \\
\mathcal{I}(\bar{e}) \leadsto \bar{x}
\end{align*}
\]

**Admissible rules.** We intend to use the rule calculus above for programming proofs of \( \mathbb{C} \text{PRL} \) judgments; for this purpose, it is very useful to be able to retreat from an incorrect rule and try a different one. This corresponds to the standard proof tactic \( \text{ORELSE} \) from LCF (Gordon et al. 1979). In our rule calculus, we render it as follows:

\[
\begin{align*}
\mathcal{I}(\bar{e}) \leadsto \bar{d} \\
\text{by rule}_1 \\
\mathcal{I}(\bar{e}_1) \leadsto x_i \\
\mathcal{I}(\bar{e}) \leadsto \bar{d} \\
\text{by rule}_1 \oplus \text{rule}_2 \\
\mathcal{I}(\bar{e}_1) \leadsto x_i \\
\mathcal{I}(\bar{e}) \leadsto \bar{d} \\
\text{by rule}_1 \oplus \text{rule}_2 \\
\mathcal{I}(\bar{e}_1) \leadsto x_i
\end{align*}
\]

Following McBride’s perspective on rules and proof tactics (McBride 1999), the expression \( \text{rule}_1 \oplus \text{rule}_2 \) can be regarded as an **admissible rule** in the sense that any judgment \( \mathcal{I} \) which can be derived in the extended calculus can also be derived in the fragment without the \( \oplus \) operator.

### 5.2 Experimental refinement logic

Here, we give a selection of refinement rules which can be used in a concrete implementation of \( \mathbb{C} \text{PRL} \); our rules resemble in large part those of the RedPRL and Nuprl systems, except that we have introduced a pervasive “twinning” of extracts; hypotheses are often strategically twinned using the equality types.\(^6\)

Whereas in Nuprl and RedPRL, it was necessary to have separate rules for constructing elements of a type, and for proving equality of elements of a type, using the twinning technique we can combine these tasks into one. Then, the usual structural equality rules all become derivable from the basic refinement-oriented theory which we have taken as primitive (see Appendix C).

**Structural rules, universes and equality**

\( \Lambda \vdash \Gamma \Rightarrow A \equiv B \) type
by equality-intro
\[
\begin{align*}
\Lambda \vdash \Gamma \Rightarrow \exists a \equiv b @ i \\
a \equiv a \ A \\
b \equiv a \ B
\end{align*}
\]

\( \Lambda \vdash \Gamma \Rightarrow A \equiv x_1 \equiv x_2 \\
x_1 \equiv x_2 \ e_1 \\
x_2 \equiv x_2 \ e_2
\]

\( \Lambda \vdash \Gamma, z : \text{Eq}_{U_i}(A; B), \Lambda \Rightarrow \exists a \equiv B @ i \\
\text{by hypothesis}[z] \\
(\text{none})
\]

\( \Lambda \vdash \Gamma, z : \text{Eq}_{U_i}(e_1; e_2), \Lambda \Rightarrow A \equiv e_1 \equiv e_2 \\
\text{by hypothesis}[z] \\
(\text{none})
\]

\( \Lambda \vdash \Gamma, z : A, \Lambda \Rightarrow e_1 \equiv e_2 \in B \\
\text{by thin}[z] \\
(\text{none})
\]

\( \Lambda \vdash \Gamma, \Lambda \Rightarrow e_1 \equiv e_2 \in B
\]

---

\(^6\)For the reviewer: we are aware of several instances of variable twinning in the literature, including Gundry (2013), but we would appreciate any pointers toward a canonical reference for this technique.
### Dependent function and pair types

| A | Γ, z : (x : A) → B[x], Δ | C ⊢ o_1[★] = o_2[★] | by x ← function-elim[z] |
| A | Γ, z : (x : A) → B[x], Δ | ⊢ A ⊢ x_1 = x_2 |
| A | Γ, z : (x : A) → B[x], Δ | ⊢ B[x_1] ⊢ x_1[x] = x_2[x] |
| A | Γ, z : (x : A) → B[x], Δ | ⊢ A ⊢ A type |

| A | Γ, z : (x : A) × B[x], Δ | z | ⊢ C[z] ⊢ x_1[fst(z), snd(z)] = x_2[fst(z), snd(z)] |
| A | Γ, z : (x : A) × B[x], Δ | ⊢ C[x, y] ⊢ x_1[x, y] = x_2[x, y] |

| A | Γ, z : (x : A) × B[x], Δ | A | ⊢ A ⊢ A type |
| A | Γ, z : (x : A) × B[x], Δ | B[x_1] | ⊢ o_1 = o_2 |
| A | Γ, z : (x : A) × B[x], Δ | ⊢ B[x] ⊢ x_1[x] = x_2[x] |

| A | Γ, z : (x : A) × B[x], Δ | A | ⊢ A ⊢ A type |
| A | Γ, z : (x : A) × B[x], Δ | B[x_1] | ⊢ o_1 = o_2 |
| A | Γ, z : (x : A) × B[x], Δ | ⊢ B[x] ⊢ x_1[x] = x_2[x] |

| A | Γ, z : (x : A) × B[x], Δ | A | ⊢ A ⊢ A type |
| A | Γ, z : (x : A) × B[x], Δ | B[x_1] | ⊢ o_1 = o_2 |
| A | Γ, z : (x : A) × B[x], Δ | ⊢ B[x] ⊢ x_1[x] = x_2[x] |

| A | Γ, z : (x : A) × B[x], Δ | A | ⊢ A ⊢ A type |
| A | Γ, z : (x : A) × B[x], Δ | B[x_1] | ⊢ o_1 = o_2 |
| A | Γ, z : (x : A) × B[x], Δ | ⊢ B[x] ⊢ x_1[x] = x_2[x] |

| A | Γ, z : (x : A) × B[x], Δ | A | ⊢ A ⊢ A type |
| A | Γ, z : (x : A) × B[x], Δ | B[x_1] | ⊢ o_1 = o_2 |
| A | Γ, z : (x : A) × B[x], Δ | ⊢ B[x] ⊢ x_1[x] = x_2[x] |

| A | Γ, z : (x : A) × B[x], Δ | A | ⊢ A ⊢ A type |
| A | Γ, z : (x : A) × B[x], Δ | B[x_1] | ⊢ o_1 = o_2 |
| A | Γ, z : (x : A) × B[x], Δ | ⊢ B[x] ⊢ x_1[x] = x_2[x] |
Clock intersection types

\[ \Lambda | \Gamma \Rightarrow \text{type } \exists \Delta \kappa. A_{\kappa}[x] \setminus A_{\kappa}[x] \atop \text{by } \kappa \leftarrow \text{clk-isect-formation} \]

\[ \Lambda, \kappa | \Gamma \Rightarrow \text{type } \exists \alpha_{\kappa}[x] \atop \text{by } \kappa \leftarrow \text{clk-isect-intro} \]

Later modality types

\[ \Lambda, \kappa | \Gamma \Rightarrow \text{type } \exists \Delta \kappa. A_{\kappa}[x] \atop \text{by later-formation}\kappa \]

\[ \Lambda, \kappa | \Gamma \Rightarrow \exists \Delta \kappa. A_{\kappa}[x] \atop \text{by later-intro} \]

It is worth noting that the elimination rules for the later modality above fulfill the essentially same role as the delayed substitutions of GDTT (Bizjak et al. 2016). The difference is that in CTT, we do not need these substitutions in our term language, and as such we do not need to be concerned with either an equational or computational account of delayed substitutions. This is a consequence of the modularity that comes from separating a type theory’s programming language from its proof system; it is also possible to experiment with
alternative elimination rules for later modalities in the proof system without needing to change the definition of CTT⊗.

### Base types

\[
\begin{align*}
\text{Type} & : \Gamma \vdash \text{bool} \equiv \text{bool} @ 0 \\
\text{Type} & : \Gamma \vdash \text{tt} \equiv \text{tt} @ 0 \\
\text{Type} & : \Gamma \vdash \text{ff} \equiv \text{ff} @ 0
\end{align*}
\]

by \text{bool-formation} \quad \text{by \text{bool-intro}} \quad \text{by \text{bool-intro}}

### Computation and auxiliary rules

\[
\begin{align*}
\text{Type} & : \Gamma \vdash \text{C} \equiv \text{n1} @ \text{n2} \\
\text{Type} & : \Gamma \vdash \text{A} \equiv \text{B} @ \text{type} \\
\text{Type} & : \Gamma \vdash \text{e1} @ \text{e2} @ \text{A} \\
\text{Type} & : \Gamma \vdash \text{A} \equiv \text{B} @ \text{type} \\
\text{Type} & : \Gamma \vdash \text{x} @ \text{n1} @ \text{n2} \\
\text{Type} & : \Gamma \vdash \text{x} @ \text{y} @ \text{type} \\
\text{Type} & : \Gamma \vdash \text{x} @ \text{y} @ \text{type} \\
\text{Type} & : \Gamma \vdash \text{x} @ \text{y} @ \text{type} \\
\text{Type} & : \Gamma \vdash \text{x} @ \text{y} @ \text{type} \\
\end{align*}
\]

by \text{compute-goal} \quad \text{by \text{compute-eq}} \quad \text{by \text{compute-eq}} \quad \text{by \text{compute-hyp}}

5.3 Program Development Notation

The refinement rules in the previous section are all written in a “small-step” style, in the sense that they each represent a single step of reasoning. Programming using these rules is possible (using the schema for derived rules), but such programs are difficult both to read and to write. We give a lightweight notation in Figure 6 for correct-by-construction programming, in the spirit of Coq’s Russell language (Sozeau 2007a,b).

As in Russell, auxiliary and well-formedness subgoals may be deferred until later; however, in our setting, this behavior is induced by simply regarding programs as notations for proof scripts, and using the \text{hole?} rule for subgoals we wish to defer. Our systematic treatment of partial construction enables us to interleave low-level tactics and rules with high-level program expressions. As such, there is no need for our programming calculus to be exhaustive, since it may be extended at-will by the user with new notations for derived rules.

Our approach can be understood as following in the footsteps of both the Epigram elaboration model (McBride 2005) as well as the Idris programming language (Brady 2013), where surface-level programs are elaborated to
programming notations for derived rules

<table>
<thead>
<tr>
<th>Step</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[ \text{let } x : A = r \text{ in } r' \triangleq x \implies \text{assert}(A; [r, r']) ]</td>
</tr>
<tr>
<td>2</td>
<td>[ \text{let } x = z(r_1) \text{ in } r_2 \triangleq x \implies \text{function-elim}[z]; [r_1, r_2] ]</td>
</tr>
<tr>
<td>3</td>
<td>[ \text{let } x : r_1 \text{ in } r_2 \triangleq x \implies \text{function-formation}; [r_1, r_2] ]</td>
</tr>
<tr>
<td>4</td>
<td>[ \text{let } x : r_1 \text{ in } r_2 \triangleq x \implies \text{pair-formation}; [r_1, r_2] ]</td>
</tr>
<tr>
<td>5</td>
<td>[ \text{let } x : r_1 \text{ in } r_2 \triangleq x \implies \text{equality-formation}; [r_1, r_2] ]</td>
</tr>
<tr>
<td>6</td>
<td>[ \text{let } x \text{ in } r_2 \triangleq x \implies \text{nat-formation} ]</td>
</tr>
<tr>
<td>7</td>
<td>[ \text{let } x : r_1 \text{ in } r_2 \triangleq x \implies \text{bool-formation} ]</td>
</tr>
</tbody>
</table>

Example 5.1 (Constructing the type of sequences). As an example, we will show how to interactively program the type of sequences of natural numbers using the rules of $\mathbb{C}^{\text{PRL}}$. We begin with the identity rule $\text{hole}?$, and will refine our proof interactively until we have induced $\mathbb{C}^{\text{PRL}}$ to produce the term $\text{fix}(X. \text{nat} \times \triangleright \kappa \ X)$.  

$$
\begin{align*}
1 & \quad \vdash \quad \text{type} \ni a_1 \equiv a_2 @ i \\
& \quad \text{by \, \text{hole}?} \\
& \quad \vdash \quad \text{type} \ni a_1 \equiv a_2 @ i \\
& \quad \text{by \{x\} \rightarrow \text{hole}?} \\
2 & \quad \vdash \quad \text{type} \ni \text{fix}(\kappa. a_1[\kappa] = \text{fix}(\kappa. a_2[\kappa]) @ i) \\
& \quad \text{by \{x\} \rightarrow \text{fix}(\kappa. a_1[\kappa] = \text{fix}(\kappa. a_2[\kappa]) @ i) \text{ by \{x\}} \\
& \quad \vdash \quad \text{type} \ni \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] \\
& \quad \text{by \{x\} \rightarrow \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] \text{ by \{x\}} \\
3 & \quad \vdash \quad \text{type} \ni \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] \\
& \quad \text{by \{x\} \rightarrow \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] \text{ by \{x\}} \\
4 & \quad \vdash \quad \text{type} \ni \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] \\
& \quad \text{by \{x\} \rightarrow \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] \text{ by \{x\}} \\
5 & \quad \vdash \quad \text{type} \ni \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] \\
& \quad \text{by \{x\} \rightarrow \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] \text{ by \{x\}} \\
& \quad \vdash \quad \text{type} \ni \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] @ i \\
& \quad \text{by \{x\} \rightarrow \text{fix}(X. \text{nat} \times \triangleright \kappa \ X) \ni a_1[\kappa] \equiv a_2[\kappa] @ i \text{ by \{x\}} \\
\end{align*}$$
The remaining subgoals are trivial and can be resolved using the rules provided. In a practical implementation of \(\mathcal{PRL}\), most such subgoals would be discharged automatically using built-in tactics, only asking for programmer-feedback when a goal falls outside the decidable fragment of the logic.

6 CONCLUSIONS AND FUTURE WORK

We have made the following contributions toward a simpler, more computational account of guarded dependent type theory:

1. We have defined a computational version of guarded dependent type theory with clocks, \(\mathcal{CTT}\), which enjoys an immediate canonicity result (Theorem 4.1).
2. We have avoided the need for any sophisticated syntax in our term language; this includes the delayed substitutions from Bizjak et al. (2016), the use of the cubical face lattice from Birkedal et al. (2016), the tick abstractions from Bahr et al. (2017).
3. Moreover, we have avoided indexing universes by clock contexts, which was necessary in Bizjak et al. (2016) to ensure the soundness of the clock irrelevance principle. In our model, this principle is validated using the fact that no clock context \(U : F_+\) is empty.
4. Finally, we have presented an implementable proof refinement logic (program logic) for \(\mathcal{CTT}\) in the style of Nuprl.

In the future, we hope to investigate the integration of \(\mathcal{CTT}\) with Computational Higher Type Theory (Angiuli et al. 2017) as part of the implementation of the RedPRL proof assistant. We would also like to explore indexing over larger ordinals than \(\omega\) to give a symmetric account to inductive types, as in Vezzosi (2015).

A SEMANTIC UNIVERSE

In this appendix, we give some further details of the semantic universe \(\mathcal{S}\).

A.1 Internal Logic and Kripke-Joyal Semantics

Using a tool called Kripke-Joyal semantics (a topos-theoretic generalization Beth/Kripke-forcing) it is possible to interpret statements in the internal language of \(\mathcal{S}\) into ordinary, external mathematical language. Each formula

PACM Progr. Lang., Vol. 1, No. 1, Article 1. Publication date: January 2018.
ϕ in the internal language stands for a subobject of some X : S⊙; the grammar of these formulae is presented below.

\[ \phi \rightarrow X : S⊙ \]
\[ \phi \lor \psi \rightarrow X : S⊙ \]
\[ \phi \land \psi \rightarrow X : S⊙ \]
\[ \phi \rightarrow \psi \rightarrow X : S⊙ \]
\[ \forall x : X. \phi(x) \rightarrow Y : S⊙ \]
\[ \exists x : X. \phi(x) \rightarrow Y : S⊙ \]
\[ \phi \rightarrow \pi (\cdot) \phi \rightarrow X \times \mathbb{K} : S⊙ \]

We will write forcing clauses U ⊩ φ(α) meaning that at world U : ⊙, the predicate φ holds of the element α : X(U). We will now proceed to propound the Kripke-Joyal forcing semantics in their entirety.

\[ U \models \phi(\alpha) \text{ presupposing } \phi \rightarrow X : S⊙, \ alpha \in X(U) \]

U ⊩ (∀κ : ∃κ. φ(α)) ⇒ φ(α)

\[ U \models (\forall x : \mathbb{K}. \phi(x, \alpha) \equiv \forall n \in \omega. (U + 1, [\partial_U, n]) \models \phi(\pi^+_U \alpha, \pi_{U+1}) \]

\[ U \models \forall x \in \mathbb{K}. \phi(x, \alpha) \equiv \forall n \in \omega. (U + 1, [\partial_U, n]) \models \phi(\pi^+_U \alpha, \pi_{U+1}) \]

A.2 Adequacy Lemmas

**Theorem A.1 (Clock irrelevance).** For φ : X : S⊙ a predicate that does not depend on the clock variable κ, we have:

U ⊩ (∀κ : ∃κ. φ(α)) ⇒ φ(α)

**Proof.** Unfolding our statement in the Kripke-Joyal semantics, this means that for all ρ : V → U, if U ⊩ ∀κ : ∃κ. φ(ρα) then V ⊩ φ(ρα). Unfolding our assumption, it suffices to exhibit some κ ∈ E(V), i.e. κ ∈ y[1](V). By definition, every world V : E+ contains at least one clock, so we are done.

**Lemma A.2.** Universal quantification of clock names can be given a simpler, equivalent forcing clause:

U ⊩ (∀x : ∃x. φ(x, α)) \equiv \forall n \in \omega. (U + 1, [\partial_U, n]) \models \phi(\pi^+_U \alpha, \pi_{U+1})

**Proof.** (⇒) This direction is trivial. (⇐) Fix ρ : V → U and κ ∈ E(V); we need to show that V ⊩ φ(ρα, κ). Observe:

\[ \forall n \in \omega. (U + 1, [\partial_U, n]) \models \phi(\pi^+_U \alpha, \pi_{U+1}) \]  
(\text{premise})
\[ \forall n \in \omega. (V + 1, [\partial_V, n]) \models \phi(\rho \pi^+_U \alpha, \pi_{V+1}) \]  
(\text{monotonicity})
\[ (V + 1, [\partial_V, \partial_V(\kappa)]) \models \phi(\rho \pi^+_U \alpha, \pi_{V+1}) \]  
(\text{instantiation})
Now observe that we can form the contraction map \( \delta_\kappa : V \rightarrow (V + 1, [\partial_V, \partial_V(\kappa)]) \), from which we have our goal by monotonicity:

\[
\begin{align*}
V \vdash \phi(\delta_\kappa \rho \pi_{\cdot \cdot \cdot \cdot} \alpha, \delta_\kappa \pi_{V+1}) \\
V \vdash \phi(\rho \alpha, \kappa)
\end{align*}
\]

(calculation)

\[\square\]

**Theorem A.3.** We can delete a later modality from under an appropriate quantification, for \( \phi \rightarrow X \times \kappa : S_\Diamond \):

\[\begin{align*}
U \vdash (\forall \kappa : \kappa. \rho_\kappa(\alpha, \kappa)) & \Rightarrow \forall \kappa : \kappa. \rho(\alpha, \kappa)
\end{align*}\]

**Proof.** It suffices to show that \( U \vdash \forall \kappa : \kappa. \rho_\kappa(\alpha, \kappa) \) implies \( U \vdash \forall \kappa : \kappa. \rho(\alpha, \kappa) \). Using Lemma A.2 as a weapon, it suffices to show that from \( \forall n \in \omega. (U + 1, [\partial_U, n]) \vdash \pi_{U+1} \rho, (\pi_{U+1}^* \alpha, \pi_{U+1}) \) we can conclude the following for any \( m \in \omega, (U + 1, [\partial_U, m]) \vdash \phi(\pi_{U+1}^* \alpha, \pi_{U+1}) \). But this follows immediately by instantiating our premise with \( n \equiv m + 1 \).

\[\square\]

**Theorem A.4 (Monotonicity).** For any predicate \( \phi \rightarrow X : S_\Diamond \), we have the following monotonicity condition:

\[\begin{align*}
U \vdash \forall \kappa : \kappa. (\rho(\alpha) \Rightarrow \rho_\kappa(\alpha))
\end{align*}\]

**Proof.** It suffices to fix \( \rho : V \rightarrow U \) and \( \kappa \in \kappa(V) \) such that \( V \vdash \phi(\rho \alpha) \), and then show that \( V \vdash \rho_\kappa \phi(\rho \alpha) \). We proceed by case on \( \partial_V(\kappa) \in \omega \):

Case \( \partial_V(\kappa) \equiv 0 \). Immediate.

Case \( \partial_V(\kappa) \equiv n + 1 \). We need to show that \( V[\kappa \mapsto n] \vdash \phi(\kappa \mapsto 1) \rho \alpha \); but this follows from monotonicity and the fact that \( V \vdash \phi(\rho \alpha) \).

\[\square\]

**Theorem A.5 (Distribution).** We have the following distributivity principle for the later modalities:

\[\begin{align*}
U \vdash \rho_\kappa(\phi(\alpha)) & \Rightarrow \psi(\alpha)) \land \rho_\kappa(\phi(\alpha)) \Rightarrow \rho_\kappa \psi(\alpha)
\end{align*}\]

**Proof.** Fixing \( \rho : V \rightarrow U \), we have to show that if \( V \vdash \rho_\kappa \phi(\rho \alpha) \Rightarrow \psi(\alpha)) \) and \( V \vdash \rho_\kappa \phi(\rho \alpha) \), then \( V \vdash \rho_\kappa \psi(\rho \alpha) \). Now, proceed by case on \( \partial_V(\rho \kappa) \in \omega \):

Case \( \partial_V(\rho \kappa) \equiv 0 \). Then we are done.

Case \( \partial_V(\rho \kappa) \equiv n + 1 \). Let \( V' \equiv V[\rho \kappa \mapsto n] \) and \( \rho' \equiv [\rho \kappa \mapsto 1] \circ \rho \). In this case, we have \( V' \vdash \phi(\rho \alpha) \Rightarrow \psi(\rho \alpha) \) and \( V' \vdash \phi(\rho \alpha) \), and we are trying to show \( V' \vdash \phi(\rho \alpha) \). Our goal follows immediately from the soundness of the modus ponens rule for implication.

\[\square\]

**Theorem A.6 (Löb induction).** We have the following Löb induction principle for the later modality:

\[\begin{align*}
U \vdash \forall \kappa : \kappa. (\rho_\kappa \phi(\alpha)) & \Rightarrow \phi(\alpha)
\end{align*}\]

**Proof.** Fixing \( \rho : V \rightarrow U \) and \( \kappa \in \kappa(V) \), such that \( V \vdash \rho_\kappa \phi(\rho \alpha) \Rightarrow \phi(\rho \alpha) \), it suffices to show that \( V \vdash \phi(\rho \alpha) \). We proceed by induction on \( \partial_V(\kappa) \in \omega \):

Case \( \partial_V(\kappa) \equiv 0 \). Then our assumption is the same as \( V \vdash \top \Rightarrow \phi(\rho \alpha) \), from which we have immediately \( V \vdash \phi(\rho \alpha) \).

Case \( \partial_V(\kappa) \equiv n + 1 \). Using modus ponens as a weapon, it suffices to show \( V \vdash \rho_\kappa \phi(\rho \alpha) \), i.e. \( V[\kappa \mapsto n] \vdash \phi([\kappa \mapsto 1] \rho \alpha) \); but this is our induction hypothesis.

\[\square\]
B SELECTED PROOFS OF RULES

In this appendix, we prove the correctness of several key rules of CTT₀ relative to the type system T₀. When manipulating terms A that depend on clocks κ, for the sake of clarity we will tend to be cautious and write A[κ].

**Formation**
\[
\frac{\Lambda, \kappa \mid \Gamma \Rightarrow A_1 \doteq A_2 \in \mathbf{U}_i}{\Lambda, \kappa \mid \Gamma \Rightarrow \joint A_1 \doteq \joint A_2 \in \mathbf{U}_i}
\]

Proof. Fix a universe level \( n : \mathbb{N} \), concrete clocks \( \Lambda \ldots \kappa : \mathbb{K} \), and context instantiations \( \gamma_1 \doteq \gamma_2 \in * \Gamma \). We need to show that \( \joint \Lambda A_1, \gamma_1 \doteq \joint \Lambda, \gamma_2 A_2 \in \mathbf{U}_n \) under the assumption that \( \gamma_1 \cdot A_1 \doteq \gamma_2 \cdot A_2 \in \mathbf{U}_n \). By the definition of the type system \( T_\omega \), our goal is the same as to say that \( c[\mathbb{U}[i]] \doteq \joint \Lambda A_1, \gamma_1 \doteq \joint \Lambda A_2, \gamma_2 \) type.

By the definition of the type system closure operator \( c[-] \), it suffices to show, \( \star c[\mathbb{U}[i]] \doteq A_1, \gamma_1 \) type, and that \( \star c([\mathbb{U}[i]] \doteq e_1 \sim e_2 \in A_1, \gamma_1) \) if and only if \( \star c([\mathbb{U}[i]] \doteq e_1 \sim e_2 \in A_2, \gamma_2) \). All of this follows essentially immediately from our premise. □

**Equality Reflection**
\[
\frac{\Lambda \mid \Gamma \Rightarrow e \in \mathbf{Eq}_A(e_1; e_2)}{\Lambda \mid \Gamma \Rightarrow e_1 \doteq e_2 \in A}
\]

Proof. Fixing concrete clocks \( \Lambda \ldots \kappa : \mathbb{K} \) and context instantiations \( \gamma_1 \doteq \gamma_2 \in * \Gamma \), we have to show that \( e_1 : \gamma_1 \doteq e_2 : \gamma_2 \in A : \gamma_1 \); we can safely assume that \( e : \gamma_1 \doteq e : \gamma_2 \in \mathbf{Eq}_A(e_1; e_2, \gamma_1 \gamma_2) \) for \( j,k \in \{1, 2\} \). By presupposition, we also have \( A : \gamma_1 \doteq A : \gamma_2 \) type. By definition of \( c[-] \), we know that \( e : \gamma_1 \leftrightarrow \star \) and \( e : \gamma_1 \doteq e : \gamma_2 \in A : \gamma_2 \). By elementary equational reasoning and the fact that member equality respects type equality, we have our goal. □

**Introduction**
\[
\frac{\Lambda, \kappa \mid \Gamma \Rightarrow e_1 \doteq e_2 \in A}{\Lambda, \kappa \mid \Gamma \Rightarrow \joint e_1 \doteq e_2 \in \mathbf{U}_i}
\]

Proof. Fixing concrete clocks \( \Lambda \ldots \kappa : \mathbb{K} \) and context instantiations \( \gamma_1 \doteq \gamma_2 \in * \Gamma \), we have to show that \( e_1 : \gamma_1 \doteq e_2 : \gamma_2 \in \mathbf{U}_i \cdot A : \gamma_1 \) under the assumption that \( e_1 : \gamma_1 \doteq e_2 : \gamma_2 \in A : \gamma_1 \). By the definition of \( c[-] \), our goal is the same as to assert \( \star c(e_1 : \gamma_1 \doteq e_2 : \gamma_2 \in A : \gamma_1) \), which holds by our assumption and Theorem A.4. □
PROOF. Fixing concrete clocks $\Lambda \ldots : \mathbb{K}$ and context instantiations $\gamma_1 \equiv \gamma_2 \in \mathbb{E} \Gamma$, we have to show that $e_1 \cdot \gamma_1 \equiv e_2 \cdot \gamma_2 \in \oplus \kappa. A[\kappa] \cdot \gamma_1$, under the assumption that for all $\kappa : \mathbb{K}$ we have $e_1 \cdot \gamma_1 \equiv e_2 \cdot \gamma_2 \in A[\kappa] \cdot \gamma_1$. This follows directly from the definition of $c[-]$. □

\[ \cap \text{ Elimination} \]
\[ \Lambda, \kappa' | \Gamma \Rightarrow e_1 \equiv e_2 \in \cap \kappa. A[\kappa] \]
\[ \Lambda, \kappa' | \Gamma \Rightarrow e_1 \equiv e_2 \in [\kappa \leftrightarrow \kappa']^{\ast}(A[\kappa]) \]

PROOF. Fixing concrete clocks $\Lambda \ldots : \mathbb{K}$ and context instantiations $\gamma_1 \equiv \gamma_2 \in \mathbb{E} \Gamma$, we have to show that $e_1 \cdot \gamma_1 \equiv e_2 \cdot \gamma_2 \in [\kappa \leftrightarrow \kappa']^{\ast}(A[\kappa] \cdot \gamma_1)$, i.e. $e_1 \cdot \gamma_1 \equiv e_2 \cdot \gamma_2 \in A[\kappa'] \cdot \gamma_1$. By the definition of $c[-]$, our assumption means that for any $\kappa : \mathbb{K}$, we have $e_1 \cdot \gamma_1 \equiv e_2 \cdot \gamma_2 \in A[\kappa] \cdot \gamma_1$. Simply choose $\kappa \equiv \kappa'$. □

\text{Clock Irrelevance}
\[
\frac{\kappa \notin \text{FreeClocks}(A) \quad \Lambda | \Gamma \Rightarrow \text{A type}}{\Lambda | \Gamma \Rightarrow \cap \kappa. \text{A} \equiv \text{A type}}
\]

PROOF. Fixing concrete clocks $\Lambda \ldots : \mathbb{K}$ and context instantiations $\gamma_1 \equiv \gamma_2 \in \mathbb{E} \Gamma$, we have to show that $(\cap \kappa. A) \cdot \gamma_1 \equiv A \cdot \gamma_2$ type under the assumption that $A \cdot \gamma_1 \equiv A \cdot \gamma_2$. Unfolding the meaning of type equality (which is extensional), this is to say that both sides of the equation are taken to the same relation by the type system $T_{\omega}$.

As a technical matter, to push inward the substitution in the left equand, we need to rename $\kappa$ to some globally fresh $\kappa'$ within the scope of the intersection to avoid capture in $\gamma_1$, i.e. $\cap \kappa'. [\kappa \leftrightarrow \kappa']^{\ast} A$. So our task is to show that $\cap \kappa'. ([\kappa \leftrightarrow \kappa']^{\ast}) A \cdot \gamma_1 \equiv A \cdot \gamma_2$ type. Because $A$ is syntactically mute in $\kappa$ by assumption, the renaming can be removed; therefore, it will suffice to show that $\cap \kappa'. A \cdot \gamma_1 \equiv A \cdot \gamma_2$ type.

Using the definition of the type system closure operator $c[-]$, this reduces to the problem of showing that the sets $(e_1, e_2) | \forall \kappa' : \mathbb{K}. e_1 \equiv e_2 \in A \cdot \gamma_2$ and $(e_1, e_2) | e_1 \equiv e_2 \in A \cdot \gamma_2$ are equal. Observe that $\kappa'$ does not occur in $e_1, \gamma_1$ or $A$ (because $\kappa'$ was chosen fresh). Therefore, by the semantic clock irrelevance principle (Theorem A1), we can delete the quantifier $\forall \kappa : \mathbb{K}$, and it remains to show that $e_1 \equiv e_2 \in A \cdot \gamma_1$ iff $e_1 \equiv e_2 \in A \cdot \gamma_2$: this follows from functionality and the fact that $\gamma_1 \equiv \gamma_2 \in \mathbb{E} \Gamma$. □

\text{\cap \text{ distribution}}
\[
\frac{\cap \text{ elimination} \quad \cap \text{ elimination}}{\cap \kappa. (x : A[k] \times B[k]) | x : \cap \kappa. A[k] \times \cap \kappa. B[k] | \text{type}}
\]

PROOF. Fixing concrete clocks $\Lambda \ldots : \mathbb{K}$ and context instantiations $\gamma_1 \equiv \gamma_2 \in \mathbb{E} \Gamma$, we have to show that the type system $T_{\omega}$ takes the types $\cap \kappa. (x : A[k] \cdot \gamma_1) \times B[k] \cdot \gamma_1$ and $(x : \cap \kappa. A[k] \cdot \gamma_1) \times (x : \cap \kappa. B[k] \cdot \gamma_2)$ to equal relations. Fixing closed terms $e_1, e_2$, this means demonstrating that the following two formulae are equivalent:

1. $(\forall \kappa : \mathbb{K}. (\text{fst}(e_1) \equiv \text{fst}(e_2) \in A[k] \cdot \gamma_1) \land (\text{snd}(e_1) \equiv \text{snd}(e_2) \in B[k] \cdot \gamma_1))$
2. $(\forall \kappa : \mathbb{K}. (\text{fst}(e_1) \equiv \text{fst}(e_2) \in A[k] \cdot \gamma_2) \land (\forall \kappa : \mathbb{K}. \text{snd}(e_1) \equiv \text{snd}(e_2) \in B[k] \cdot \gamma_2))$

Because universal quantification commutes with conjunction, it suffices to show that the following are equivalent, fixing $\kappa : \mathbb{K}$:

1. $(\text{fst}(e_1) \equiv \text{fst}(e_2) \in A[k] \cdot \gamma_1) \land (\text{snd}(e_1) \equiv \text{snd}(e_2) \in B[k] \cdot \gamma_1)$
2. $(\text{fst}(e_1) \equiv \text{fst}(e_2) \in A[k] \cdot \gamma_2) \land (\text{snd}(e_1) \equiv \text{snd}(e_2) \in B[k] \cdot \gamma_2)$

PACM Progr. Lang., Vol. 1, No. 1, Article 1. Publication date: January 2018.
This amounts to the type functionality of $A[\kappa]$ and $B[\kappa]$ with respect to $\Gamma$, which is exactly the content of our two premises.

C DERIVABILITY OF STRUCTURAL EQUALITY REFINEMENT RULES

We have claimed that the usual structural equality rules for the refinement judgment $\Lambda \vdash \Gamma \implies e_1 \equiv e_2 \in A$ are derivable from the rules for program refinement, which target the judgment $\Lambda \vdash \Gamma \implies A \ni x_1 \equiv x_2$. We will give a few examples.

C.1 Structural Equality for Introduction Forms

The derivation of structural equality rules for introduction forms all follow the same pattern; as an example, we will derive the structural equality rule for lambda expressions, culminating in the definition of a tactic `lambda-equality`:

$$\Lambda \vdash \Gamma \implies \lambda x. e_1 \equiv \lambda x. e_2 \in (x : A) \rightarrow B$$

**by lambda-equality**

$$\Lambda \vdash \Gamma, x : A \implies e_1 \equiv e_2 \in B$$

$$\Lambda \vdash \Gamma \implies A \equiv A \text{ type}$$

To understand how the derivation works, we will walk through it step-by-step.

1. $\Lambda \vdash \Gamma \implies \lambda x. e_1 \equiv \lambda x. e_2 \in (x : A) \rightarrow B$
   **by hole?**
   $$\Lambda \vdash \Gamma \implies \lambda x. e_1 \equiv \lambda x. e_2 \in (x : A) \rightarrow B$$

First, we use the equality judgment’s introduction rule to enter program refinement mode. This induces two auxiliary alpha equivalence subgoals, which we will solve at the very end once each $x_i$ has been instantiated in the course of the proof.

2. $\Lambda \vdash \Gamma \implies \lambda x. e_1 \equiv \lambda x. e_2 \in (x : A) \rightarrow B$
   **by equality-intro; hole?, hole?, hole?**

In fact, we can use the `try{r}` tactical to automatically resolve the side conditions as soon as $x_i$ have been instantiated:

2'. $\Lambda \vdash \Gamma \implies \lambda x. e_1 \equiv \lambda x. e_2 \in (x : A) \rightarrow B$
   **by equality-intro; hole?, try{trivial}, try{trivial}**

Next, we introduce a function, which causes $x_i$ to be replaced with $\lambda x. x_i[x]$ in the remainder of the subgoals.
The next step is to assert a new hypothesis $p$ of type $B$, which will induce two subgoals: a subgoal to establish that $B$ is inhabited, and a subgoal to use this fact. Notice how the assertion rule does not naively hypothesize an arbitrary element of the type $B$, but rather hypothesizes that the actual twin witnesses which we are constructing in the other subgoal are equal elements of $B$. This allows us to carefully control the extraction process to ensure that precisely the correct term ends up in the output.

Next, we need to inhabit $B$ by twin elements $e_1, e_2$, which is exactly the function of the rule $\text{exact}(e_1, e_2)$; this induces the corresponding equality subgoal, which is a premise of the derived rule we intended to justify.

It remains only to use our hypothesis $p$; once we do so, by filling in the remaining hole, it causes $x_i[x, p]$ to be replaced by $e_1$, which enables the $\text{try} \{ \text{trivial} \}$ tactics to fire and resolve our remaining subgoals.
C.2 Structural Equality for Elimination Forms

The structural equality of elimination forms follows a slightly different pattern; we will derive the equality of function applications as a paradigmatic example.

In Nuprl-style systems, the structural equality rule that proves $\Lambda \vdash \Delta \equiv \Delta \Rightarrow f_1(x) \equiv f_2(x) \in B[e_1]$ requires as an argument an appropriate dependent function type $(x : A) \rightarrow B[e]$, leading to a family of derived rules along the following lines:

$$
\Lambda \vdash \Delta \equiv \Delta \Rightarrow f_1(x) \equiv f_2(x) \in B[e_1]
$$

by `ap-equality(A, x.B[e])`

$$
\Lambda \vdash \Delta \equiv \Delta \Rightarrow f_1 \equiv f_2 \in (x : A) \rightarrow B[x]
\Lambda \vdash \Delta \equiv \Delta \Rightarrow e_1 \equiv e_2 \in A
$$

Using a type synthesis heuristic, it is usually easy to determine the specific parameters $A, x.B[x]$ to pass to the above rule. We will regard that as a separable issue and proceed with the derivation of the parameterized equality structural equality rule above.

As with our previous example, we use the `equality-intro` rule, together with some tactics to resolve the resulting alpha equivalence subgoals as soon as they become sufficiently instantiated.
We are now trying to construct twin elements of $B[e_1]$ which are both function applications; naïvely, we may wish to use the function elimination rule, but recall that this only applies to a hypothesis, whereas $f_i$ may be an arbitrary program rather than a variable. Luckily, we can use the assertion rule again to introduce a suitable variable $p$ of functional type into our context.

$$
\lambda \vdash \square \Rightarrow f_1(e_1) \equiv f_2(e_2) \in B[e_1]
$$

by equality-intro;

\begin{align*}
\lambda \vdash (x : A) \rightarrow B[x] \ni \eta_1 \equiv \eta_2 & \\
\lambda, \Gamma, p : \text{Eq}(x : A) \rightarrow B[x](\eta_1; \eta_2) \Rightarrow B[e_1] \ni x_1[p] \equiv x_2[p] & \\
x_1[\star] \equiv_{\alpha} f_1(e_1) & \\
x_2[\star] \equiv_{\alpha} f_2(e_2) &
\end{align*}

Next, we specify that $p$ shall be implemented by exactly $f_1, f_2$, which induces the exact equality subgoal that we intend to be induced by our derived rule.

$$
\lambda \vdash \square \Rightarrow f_1(e_1) \equiv f_2(e_2) \in B[e_1]
$$

by equality-intro;

\begin{align*}
\lambda \vdash (x : A) \rightarrow B[x] & \\
\lambda, \Gamma, p : \text{Eq}(x : A) \rightarrow B[x](f_1; f_2) \Rightarrow B[e_1] \ni x_1[p] \equiv x_2[p] & \\
x_1[\star] \equiv_{\alpha} f_1(e_1) & \\
x_2[\star] \equiv_{\alpha} f_2(e_2) &
\end{align*}

Now, we have a hypothesis $p : \text{Eq}(x : A) \rightarrow B[x](f_1; f_2)$, but we want to turn it into a hypothesis of type $(x : A) \rightarrow B[x]$ so that we can apply the function elimination rule to it. Recall that the reason for assertions introducing such equality hypotheses rather than raw hypotheses is that it is often important to know the exact identity of the asserted element when using it, as opposed to constructing the goal on the basis of an arbitrary element. Luckily, the Eq-peel rule is precisely what we need: it will replace our hypothesis, and then fill in $p$ with the equated elements in the remainder of the subgoals.
Now we are in a position to apply the function elimination rule to the hypothesized function \( p \), which induces a subgoal of type \( A \) and a twin hypothesis of the output type.

To fill in the argument subgoal we use \( \text{exact}(e_1, e_2) \), which gives us our second desired equality subgoal.
What remains is to exhibit \( B[e_1] \), which we have already hypothesized as a result of applying the function elimination rule. So, we fill in our hole with the twin hypothesis \( q \).

\[
\begin{align*}
8 \quad & \quad \Lambda \mid \Gamma \Rightarrow f_1(e_1) \doteq f_2(e_2) \in B[e_1] \\
& \quad \text{by equality-intro;} \\
& \quad \begin{aligned}
\text{let } p : (x : A) \rightarrow B[x] = \text{exact}(f_1, f_2) \text{ in} \\
\text{Eq-peel}[p]; \\
\text{let } q = p(\text{exact}(e_1, e_2)) \text{ in } q, \\
\text{try}\{\text{trivial}\}, \\
\text{try}\{\text{trivial}\}
\end{aligned}
\end{align*}
\]

Now, the derived rule is in almost perfect condition, except that we have an undesirable hypothesis \( p \) in our second subgoal; this can be removed by adding a call to the rule \( \text{thin}[p] \).

\[
\begin{align*}
9 \quad & \quad \Lambda \mid \Gamma \Rightarrow f_1(e_1) \doteq f_2(e_2) \in B[e_1] \\
& \quad \text{by equality-intro;} \\
& \quad \begin{aligned}
\text{let } p : (x : A) \rightarrow B[x] = \text{exact}(f_1, f_2) \text{ in} \\
\text{Eq-peel}[p]; \\
\text{let } q = p(\text{exact}(e_1, e_2); \left[\text{thin}[p]\right]) \text{ in } q, \\
\text{try}\{\text{trivial}\}, \\
\text{try}\{\text{trivial}\}
\end{aligned}
\end{align*}
\]

\( \square \)

ACKNOWLEDGMENTS

We are thankful to Aleš Bizjak, Lars Birkedal, Bas Spitters, Adrien Guatto, Carlo Angiuli, Pieter Hostra, Jonas Frey, Sam Staton and Daniel Gratzer 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.

REFERENCES


PACM Progr. Lang., Vol. 1, No. 1, Article 1. Publication date: January 2018.


