Senior Thesis

The Next 700 Failed Step-Index-Free Logical Relations

Daniel Gratzer

supervised by
Professor Karl Crary

May 4, 2018

Abstract

An important topic in programming languages is the study of program equivalence. This is done typically with the construction of a relational denotational model or a syntactic analogue, called a logical relation. Logical relations have proven to be an effective tool for analyzing programs and lending formal weight to ideas like data abstraction and information hiding.

A central difficulty with logical relations is their fragility; it has proven to be a challenge to scale logical relations to more realistic languages. A common technique for accomplishing this is *step-indexing.* Step-indexing may be an effective tool for defining logical relations but it results in often frustrating technical details and limitations. Replacing it with more traditional logical relations is desirable but so far has only been achieved for recursive types. In this work we consider extending traditional logical relations to higher-order references, a common feature in modern languages.

The central challenge with constructing a logical relation for higher-order references is that it must be a Kripke logical relation. The construction of the Kripke worlds has been a persistent challenge because they do not exist as mere sets. The semantic types must be indexed by a Kripke world and the Kripke world must mention those semantic types, a recursive equation which has no solutions naively. This can be solved by using step-indexing to navigate the recursive equation but other techniques may be possible.

Three methods are explored in this thesis: a domain theoretic approach, a naive application syntactic minimal invariance, and an application of syntactic minimal invariance to step-indexing. None are sufficient to solve the problem. This demonstrates the difficulty of expressing the recursive structure of higher-order references.

1 Introduction

In the study of programming languages a great number of important questions hinge on the equality of programs.

- The verification of a compiler pass is nothing but a question of equality of the naive program and its optimized version.
- An optimized data structure may be shown to be correct relative to a much simpler reference solution.
- Much of the research in dependent type theory centers around what terms should be judged equal.

Given the role that equality plays in programming languages it is unsurprising that a great variety of mathematical tools have been developed to study it.

Before anything can be said about tools for analyzing equality, however, a precise characterization of equality must be given. There are many reasonable notions of equality between programs and all of them are suitable for different circumstances. For instance, one can compare programs up to renaming of variables, α-equivalence, in compilers and other applications where it is important that no information is lost. In type-checkers for dependently typed languages, it is natural to be more flexible and regard programs as the same if they share a normal form, some sort of β-equivalence at least. Here we are interested in studying program behavior and verifying properties about it so an even more relaxed notion of equality is called for: *contextual equivalence*.

Contextual equivalence is formally defined later (see Definition 2.3) but informally contextual equivalence of two programs, $e_1 \cong e_2$, expresses that e_1 and e_2 are internally indistinguishable. This means that there is no program of base type containing e_1 so that if e_1 is replaced with e_2 then the program returns a different result. This notion of equivalence is appealing because there is no aspect of a program at base type we care about other than what it computes to. Contextual equivalence allows us to ignore unimportant differences in the precise way that the answer of a program is computed and focus instead on the answer itself. This means that two implementations of a data structure, for instance, are contextually equivalent even if they differ greatly in complexity or efficiency.

For all the appeal of contextual equivalence, it is very difficult to establish any instances of it. In order to show that $e_1 \cong e_2$ it is necessary to quantify over all possible programs using e_1 and e_2 and reason about their behavior. This includes programs which do not really make use of e_1 or e_2 such as $(\lambda x{:}\mathsf{unit} \to \tau.\, 1)\ (\lambda x{:}\mathsf{unit}.\, e_1)$. This lack of constraints on how e_1 or e_2 is used is precisely what makes contextual equivalence so useful but it makes even the most basic proofs involved affairs. In order to compensate for this, a variety of tools have been specialized to simplify the process of establishing when $e_1 \cong e_2$.

Broadly speaking, there are two classes of these tools. One may consider denotational approaches, where in the equality of programs is reduced to the question of the equality of normal mathematical objects. This line of study begins with Dana Scott's investigations of the lambda calculus [49]. Such tools have been immensely effective when they may be applied but they are difficult to use. Complex programming languages often make use of extremely sophisticated mathematical objects and using the model requires understanding them. This has meant that denotational semantics is traditionally out of reach for the verification of programs by an average programmer or anyone besides a domain expert.

On the other hand, there are syntactic approaches to equality. Some of these date back to the original study of the lambda calculus and its reduction properties. Syntactic tools tend to be simpler to use, relying only one elementary mathematics and an understanding of the syntax itself. For equality, the tool of choice when working syntactically is a logical relation [3, 4, 6, 7, 24, 25, 31, 44, 53]. Dating back to Tait [53] logical relations generalize homomorphisms and can be thought of as *structure preserving relations*[1]. Logical relations are used for a wide variety of tasks but the most important one for us is as a method for demonstrating contextual equivalences. The study of logical relations has given rise to a powerful generalization of the theory of data abstraction called parametricity [47]. Parametric reasoning has been so important that there have even been attempts to reconstruct it in a denotational sense [1, 10, 13, 26, 37]. This work is important for crystallizing the connections between naturality and parametricity as well as cementing parametricity in the broader tradition of logical relations. At the present time this program is still incomplete however. Hermida et al. [34] is a good summary of the state of the art and the fundamental challenges in 2014.

To be precise, a logical relation for us is a family of binary relations indexed by the types of the language, $(R_\tau)_{\tau \in \text{Type}}$. It is constructed by induction over the types indexing it so that $R_{\tau_1 \to \tau_2}$ is defined in terms of R_{τ_1} and R_{τ_2}. For each τ, the following property is expected to hold:

$$(e_1, e_2) \in R_\tau \implies e_1, e_2 : \tau \wedge e_1 \cong e_2$$

This property expresses the soundness of the logical relation with respect to contextual equivalence. The reverse property, completeness, is desirable but often unachievable[2]. Completeness is usually unnecessary for establishing certain concrete equivalences so in this context it is less important.

Finally, an important property of logical relations which is difficult to capture formally is that it is much easier to prove that $(e_1, e_2) \in R_\tau$ than to directly show that $e_1 \cong e_2$. Typically, R_τ is meant to capture the logical action of τ; it expresses the precise set of observations possible to make of expressions of type τ without all the duplication of contextual equivalence. For instance, in order

[1]This viewpoint is more liberating than the traditional explanation that they are a type-indexed inductively defined family of relations. Logical relations arise for general mathematical structure. A programming language is merely one example of such structure.

[2]Instead a variety of techniques for *forcing* completeness to hold are employed. These amount to adding all the missing identifications to the logical relation. While theoretically desirable, this is useless for the problem of actually establishing an equivalence.

to show that $(e_1, e_2) \in R_{\tau_1 \to \tau_2}$ it suffices to show the following.

$$\forall (a_1, a_2) \in R_{\tau_1}.\ (e_1\ a_1, e_2\ a_2) \in R_{\tau_2}$$

This is significantly easier than showing that any program using two functions must behave the same. This definition expresses the fact that the only real way to use a function is to apply it.

Given the obvious appeal of having a logical relation defined for a language, it would seem that the construction of a logical relation characterizing equality is a natural first step in the study of a new programming language. The central stumbling block to this goal is that logical relations are difficult to extend to new programming languages and especially to new types and computational effects.

To see where trouble might arise, consider a language with polymorphic types: $\forall \alpha. \tau$. What should the definition of $R_{\forall \alpha. \tau}$ be? Phrased differently, when are two polymorphic functions indistinguishable? One might expect a definition like the following:

$$(e_1, e_2) \in R_{\forall \alpha. \tau} \triangleq \forall \tau'.\ (e_1[\tau'], e_2[\tau']) \in R_{[\tau'/\alpha]\tau}$$

This does correctly characterize contextual equivalence but it is ill-suited as a definition in a logical relation because it is not well-founded. A logical relation is constructed by induction on the types and with impredicative polymorphism there is absolutely no reason why τ' should be prior than $\forall \alpha. \tau$ in the ordering used for induction. This is not a minor issue, defining a correct logical relation for a language with impredicative polymorphism requires the method of candidates [30, 31]. This situation has been a recurring issue: features in programming languages tend to have some recursive structure which prevents them from easily fitting into the inductive definition of a logical relation. For many features, clever constructions have been found which circumvent well-foundedness issues: for instance, for parametric polymorphism [30, 31], first-order state [44], simple exceptions, and recursive types [12, 22]. Despite this work, logical relations are still a long way away from being able to cope with many full programming languages. The state of the art for dealing with features like higher-order references, nontrivial control passing, or concurrency is to use step-indexing.

Step-indexing is a technique first proposed by Appel and McAllester [6]. It is based on a simple but ingenious idea: if your logical relation is not well-defined by induction on the type just add a number which decreases and induct on that. This idea means that a logical relation is no longer a type-indexed relation but a relation indexed by a number (called a *step*) and a type, $(R^i_\tau)_{i,\tau}$.

The meaning of two programs related at step i and type τ is that they are indistinguishable at type τ for i steps. Two programs are related for i steps intuitively if it will take at least i steps in order to make an observation which distinguishes them. For instance, if a program e_1 runs to true in 10 steps while e_2 runs to false in 20, it takes 20 steps to distinguish them. Semantically, this idea is crystallized in models of programming languages in metric spaces [27].

This idea of steps has proven to be incredibly robust and easily extend to many different settings [3, 4, 6, 7, 17, 24, 25, 52, 54]. There is a price to be paid for this extra flexibility. Firstly, steps now pervade the definition of the logical relation and any proof now must contain pointless bookkeeping activity to keep track of them. Secondly, the evaluation behavior of the programs under consideration has begun to matter again. The goal of contextual equivalence was to erase it from consideration. Now there are instances where in order to use a logical relation a program must be modified with spurious NO-OP statements in order to make it take extra steps, a clear violation of this principle. These issues are even present when just using the logical relation to validate contextual equivalences [52]. In practice, this has meant that while step-indexing is incredibly widely used it is almost universally disliked.[3]

The structure of this thesis is as follows. In Section 2 we describe the ML-like language that will serve as target for our logical relations. In Section 3 we construct a step-indexed logical relation for this language, paying attention to its shortcomings. In Section 4 we describe the first failed step-index-free logical relation: the most naïve approach centered around using domain theory. In Sections 5 and 6 we explore a variety of approaches at step-index-free logical relations for languages powerful enough to encode general references, largely built around (guarded) recursive kinds.

2 An ML-like Language with General References

In order to make the discussion of logical relations more concrete, a particular language is necessary. In this section we develop a core calculus suitable for studying the effects of general references on reasoning.

The language under consideration here is heavily influenced by the Modernized Algol discussed by Harper [32]. It features a syntactic separation between commands and expressions. Expressions are unable to depend on in any way on the heap. Commands may modify the heap using assignables, a mutable "variable" that clarifies the binding structure (nominal rather than substitutive). A crucial component of the system is a modality for internalizing and suspending commands to treat them as expressions. Arguably this language is heavier-weight than the ML-like languages usually discussed; normally languages considered allow imperative computation anywhere in the program. A language with isolated commands has the advantage that handling state is now isolated in the logical action of the command modality. In traditional presentations this complexity is is not littered throughout the full logical relation. Indeed, the definitions of the logical relation at $\to$ or $\forall$ are almost completely unchanged from System F for our logical relation.

The syntax of our language has three sorts: commands, expressions, and

[3]Citation: I walked around at POPL and the five people I asked seemed like REALLY bummed about it.

types defined by the following grammar.

$$\begin{array}{rcl}
\tau & ::= & \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau \mid \mathsf{cmd}(\tau) \\
c & ::= & \mathsf{ret}(e) \mid \mathsf{get}[\alpha] \mid \mathsf{set}[\alpha](e) \\
 & \mid & \mathsf{dcl}\ \alpha := e\ \mathsf{in}\ c \mid \mathsf{bnd}\ x \leftarrow e;\ c \\
e & ::= & x \mid e\ e \mid \lambda x{:}\tau.\, e \mid \Lambda \alpha.\, e \mid \mathsf{cmd}(e)
\end{array}$$

Only one point of this syntax must be clarified, which is the unfortunate coincidence of αs. In an attempt to maintain consistency with the standard literature on System F and Harper's Modernized Algol α here refers either to an assignable (a symbol) or a type variable.

A crucial point of this language is that assignables are *not* variables. Assignables are bound by the operators of our language and they do α-vary as a variable might but you can not substitute for an assignable. This justifies comparing bound symbols for equality; they are less placeholders and more pointers to binders. It is nonsensical to talk about a rule like the following.

$$\frac{}{\mathsf{dcl}\ \alpha := v\ \mathsf{in}\ c \mapsto [v/\alpha]c}$$

This is similar to the confusion that many programmers have when discussing languages with "variables": C-like languages do not permit substitution because they do not possess variables but rather assignables. In C a rule like the above is clearly false and leads to statements such as `1 = 2`. One can understand the difference between $\mathsf{bnd}\ x \leftarrow e;\ c$ and $\mathsf{dcl}\ x := e\ \mathsf{in}\ c$ as the difference between *variables*, defined through substitution, and *assignables* which are defined by the site in which they are bound. The former is like a let-binding while the latter is closer to a declaration in C. In our language this separation exists which is why the operators for reading and writing a mutable cell are indexed by symbol rather than taking an arbitrary term. There is much to be said on the subject of symbols and, more generally, nominal binding but it is sadly out-of-scope for this discussion. The interested reader is referred to [43].

The static semantics of the language are divided into three judgments, the first of which is the judgment ensuring a type is a well-formed. Informally, a type is well formed in a context Δ if Δ contains all the free variables of the type.

$$\boxed{\Delta \vdash \tau : \mathsf{T}}$$

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha : \mathsf{T}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T} \qquad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \to \tau_2 : \mathsf{T}} \qquad \frac{\Delta, \alpha \vdash \tau : \mathsf{T}}{\Delta \vdash \forall \alpha.\, \tau : \mathsf{T}} \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathsf{cmd}(\tau) : \mathsf{T}}$$

In order to explain the statics of expressions and commands two judgments are necessary and they must depend on each other. This dependence stems from $\mathsf{cmd}(-)$ which internalizes the command judgment. First the expression judgment is given, it is completely standard except that it must also be fibered over a specification of the available assignables. This extra context is necessary in order to make sense of the binding done in dcl.

$$\boxed{\Delta; \Gamma \vdash_\Sigma e : \tau}$$

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Delta; \Gamma, x : \tau_1 \vdash_\Sigma e : \tau_2}{\Delta; \Gamma \vdash_\Sigma \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}$$

$$\frac{\Delta; \Gamma \vdash_\Sigma e_1 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash_\Sigma e_2 : \tau_1}{\Delta; \Gamma \vdash_\Sigma e_1\ e_2 : \tau_2} \qquad \frac{\Delta, \alpha; \Gamma \vdash_\Sigma e : \tau}{\Delta; \Gamma \vdash_\Sigma \Lambda\alpha.\, e : \forall\alpha.\, \tau}$$

$$\frac{\Delta; \Gamma \vdash_\Sigma e : \forall\alpha.\, \tau_1 \qquad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta; \Gamma \vdash_\Sigma e[\tau_2] : [\tau_2/\alpha]\tau_1} \qquad \frac{\Delta; \Gamma \vdash_\Sigma m \div \tau}{\Delta; \Gamma \vdash_\Sigma e : \tau}$$

These are the rules of System F with the exception of the final one. This makes use of the judgment for commands, where $m \div \tau$ signifies that m is a command which (when executed on the appropriate heap) will run to a $\mathsf{ret}(v)$ for some $v : \tau$. It is worth noting that Σ in this judgment is unimportant except in this rule as well. It is only necessary to track the available assignables because without this information it is not possible to determine if a command is well-typed. The rules for commands are as follows.

$$\frac{\Delta; \Gamma \vdash_\Sigma e : \tau}{\Delta; \Gamma \vdash_\Sigma \mathsf{ret}(e) \div \tau} \qquad \frac{\alpha \div \tau \in \Sigma}{\Delta; \Gamma \vdash_\Sigma \mathsf{get}[\alpha] \div \tau} \qquad \frac{\alpha \div \tau \in \Sigma \qquad \Delta; \Gamma \vdash_\Sigma e : \tau}{\Delta; \Gamma \vdash_\Sigma \mathsf{set}[\alpha](e) \div \tau}$$

$$\frac{\Delta; \Gamma \vdash_\Sigma e : \tau_2 \qquad \Delta; \Gamma \vdash_{\Sigma, \alpha : \tau_2} m \div \tau_1}{\Delta; \Gamma \vdash_\Sigma \mathsf{dcl}\ \alpha := e\ \mathsf{in}\ m \div \tau_1}$$

$$\frac{\Delta; \Gamma \vdash_\Sigma e : \mathsf{cmd}(\tau_1) \qquad \Delta; \Gamma, x : \tau_1 \vdash_\Sigma m \div \tau_2}{\Delta; \Gamma \vdash_\Sigma \mathsf{bnd}\ x \leftarrow e;\ m \div \tau_2}$$

The operational semantics of this language can be presented in two distinct ways differing in the way that the allocation of a fresh heap cell is done. In classical literature, an expression is evaluated using traditional small step semantics. Commands in this framework are evaluated by a small step semantics as well but the relation is between configurations, not just commands. A configuration is a pair (m, h), a command and the heap it is running on. In Harper [32] the binding structure of symbols is preserved in the operational semantics so that a running command is represented as a configuration $\nu\Sigma.\ (m, h)$ where Σ is a collection of symbols bound in both m and h.

The advantage of this presentation is that it eliminates the choice of how to pick a fresh symbol during allocation. Allocated assignables are bound and so they automatically α-vary to avoid collisions with other assignables. In the more traditional set up the stepping relation is nondeterministic, allowing for any fresh location to be chosen. This nondeterminism complicates many metatheoretic properties and obscures what is really happening since the language is fundamentally deterministic! The formal presentation has just obscured the precise meaning of a construct and approximated it with nondeterminism. For the problems discussed in this thesis, however, this advantage is not significant and it proves to be (while not impossible) onerous to convert from semantic objects to Σ when running programs. For concision and expository purposes the traditional operational semantics are chosen.

$$\boxed{v \ \mathsf{value}}$$

$$\frac{}{\lambda x{:}\tau.\, e \ \mathsf{value}} \qquad \frac{}{\Lambda\alpha.\, e \ \mathsf{value}} \qquad \frac{}{\mathsf{cmd}(m) \ \mathsf{value}}$$

$$\boxed{e \mapsto e'}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{v \ \mathsf{value} \qquad e_2 \mapsto e_2'}{v\ e_2 \mapsto v\ e_2'} \qquad \frac{v \ \mathsf{value}}{(\lambda x{:}\tau.\, e)\ v \mapsto [v/x]e}$$

$$\frac{e \mapsto^* e'}{e[c] \mapsto^* e'[c]} \qquad \frac{}{(\Lambda\alpha.\, e)\ c \mapsto [c/\alpha]e}$$

$$\boxed{(m, h) \ \mathsf{final}}$$

$$\frac{v \ \mathsf{value}}{(\mathsf{ret}(v), h) \ \mathsf{final}}$$

$$\boxed{(m, h) \mapsto^* (m', h')}$$

$$\frac{e \mapsto e'}{(\mathsf{ret}(e), h) \mapsto (\mathsf{ret}(e'), h)} \qquad \frac{e \mapsto e'}{(\mathsf{bnd}\ x \leftarrow e;\ m, h) \mapsto (\mathsf{bnd}\ x \leftarrow e';\ m, h)}$$

$$\frac{(m_1, h) \mapsto (m_1', h')}{(\mathsf{bnd}\ x \leftarrow \mathsf{cmd}(m_1);\ m_2, h) \mapsto (\mathsf{bnd}\ x \leftarrow \mathsf{cmd}(m_1');\ m_2, h')}$$

$$\frac{v \ \mathsf{value}}{(\mathsf{bnd}\ x \leftarrow \mathsf{cmd}(\mathsf{ret}(v));\ m, h) \mapsto ([v/x]m, h)}$$

$$\frac{e \mapsto e'}{(\mathsf{dcl}\ \alpha := e \ \mathsf{in}\ m, h) \mapsto (\mathsf{dcl}\ \alpha := e' \ \mathsf{in}\ m, h)}$$

$$\frac{v \ \mathsf{value} \qquad \alpha \,\#\, h}{(\mathsf{dcl}\ \alpha := v \ \mathsf{in}\ m, h) \mapsto (m, h[\alpha \mapsto v])} \qquad \frac{}{(\mathsf{get}[\alpha], h) \mapsto^* (\mathsf{ret}(h(\alpha)), h)}$$

$$\frac{e \mapsto e'}{(\mathsf{set}[\alpha](e), h) \mapsto^* (\mathsf{set}[\alpha](e'), h)} \qquad \frac{v \ \mathsf{value}}{(\mathsf{set}[\alpha](v), h) \mapsto^* (\mathsf{ret}(v), h[\alpha \mapsto v])}$$

This gives a precise account of the language under consideration. Still undefined, however, is the right notion of equivalence. It was informally sketched in the introduction as the ability to replace one program with the other in an arbitrary context. It is now possible to make this definition formal. First, a context is a term with a distinguished hole in it, written $\square$. They are generated from the following grammar:

$$\begin{array}{lll}
C & ::= & \square \mid C\ e \mid e\ C \mid \lambda x{:}\tau.\, C \mid \Lambda\alpha.\, C \mid C[c] \mid \mathsf{cmd}(C_m) \\
C_m & ::= & \mathsf{ret}(C) \mid \mathsf{bnd}\ x \leftarrow C;\ m \mid \mathsf{bnd}\ x \leftarrow e;\ C_m \\
 & \mid & \mathsf{dcl}\ \alpha := C\ \mathsf{in}\ m \mid \mathsf{dcl}\ \alpha := e\ \mathsf{in}\ C_m \mid \mathsf{set}[\alpha](C) \\
C^m & ::= & C^m\ e \mid e\ C^m \mid \lambda x{:}\tau.\, C^m \mid \Lambda\alpha.\, C^m \mid C^m[c] \mid \mathsf{cmd}(C^m_m) \\
C^m_m & ::= & \square \mid \mathsf{ret}(C^m) \mid \mathsf{bnd}\ x \leftarrow C^m;\ m \mid \mathsf{bnd}\ x \leftarrow e;\ C^m_m \\
 & \mid & \mathsf{dcl}\ \alpha := C^m\ \mathsf{in}\ m \mid \mathsf{dcl}\ \alpha := e\ \mathsf{in}\ C^m_m \mid \mathsf{set}[\alpha](C^m)
\end{array}$$

There are four varieties of contexts, representing that an {expression, command} may be put into another {expression, command}. C is the class of contexts inserting expressions into expressions while C_m allows for inserting expressions into commands. C^m and C^m_m allow for inserting commands into expressions and commands into commands respectively.

All of these come equipped with a natural operation replacing $\square$ with an expression or command, denoted $C[e]$ or $C[m]$. Notice that crucially this operation allows variable capture; e is allowed to make use of the variables bound in C. This is important for allowing contextual equivalence to work on open terms.

A small point of interest is that having two sorts which may mention each other leads to four different varieties of contexts. This quadratic growth continues which makes contextual equivalence difficult to define for complex languages. This has been addressed in Crary [21] by defining contextual equivalence as the unique relation satisfying Theorem 2.4. Finally, contexts can be equipped with a notion of typing. This typing relation for a context representing an expression with an expression hole, C, expresses that when filled with a program $\Delta; \Gamma \vdash_\Sigma e : \tau$ then $\Delta'; \Gamma' \vdash_{\Sigma'} C[e] : \tau'$. This typing relation is unsurprising and therefore omitted. This judgment is written as $C : (\Delta; \Gamma \vdash_\Sigma \tau) \rightsquigarrow (\Delta'; \Gamma' \vdash_{\Sigma'} \tau')$ if C sends an object of type τ in context $\Delta; \Gamma; \Sigma$ to an object of type τ' in context $\Delta'; \Gamma'; \Sigma'$.

In order to define contextual equivalence, two supplementary definitions are required.

Definition 2.1. *A heap, h matches Σ, written $h : \Sigma$, if for all $\alpha \div \tau \in \Sigma$, $\cdot; \cdot \vdash_\Sigma h(l) : \tau$.*

Definition 2.2. *Two programs are said to be Kleene equal at Σ, written $e_1 \simeq_\Sigma e_2$ if for all $h : \Sigma$ and m_1, m_2 so that $e_i \mapsto^* \mathsf{cmd}(m_i)$ then $(m_1, h) \Downarrow$ and $(m_2, h) \Downarrow$ or $(m_1, h) \Uparrow$ and $(m_2, h) \Uparrow$.*

This definition of Kleene equality is slightly non-standard in two ways: it is defined for expressions which compute to commands and it only checks if the resulting commands coterminate, not that they result in the same value. These deviations are natural in this language because without them Kleene equivalence is unhelpful. There is no base observable type in this language and so there is no observation to be made besides termination. Furthermore, it is not hard to prove that expressions always terminate (they behave as System F with a new base type) so *no* interesting observations may be made of just expressions.

We will also make use of the notation $(m_1, h_1) \simeq (m_2, h_2)$ to signify just the final part of this definition. Finally, we can lift Kleene equality to a definition of contextual equivalence.

Definition 2.3. *Two programs are contextually equivalent,* $\Delta; \Gamma \vdash_\Sigma e_1 \cong e_2 : \tau$ *if for all contexts* $C : (\Delta; \Gamma \vdash_\Sigma \tau) \rightsquigarrow (\cdot; \cdot \vdash_{\Sigma'} \mathsf{cmd}(\tau'))$, $C[e_1] \simeq_{\Sigma'} C[e_2]$.

Similarly, two commands are said to be contextually equivalent if $C^m : (\Delta; \Gamma \vdash_\Sigma \tau) \rightsquigarrow (\cdot; \cdot \vdash_{\Sigma'} \mathsf{cmd}(\tau'))$, $C^m[m_1] \simeq_{\Sigma'} C^m[m_2]$.

This definition formalizes our earlier intuitions and it makes obvious the issues that were mentioned with contextual equivalence previously. In order to establish contextual equivalence we must write a proof which handles an arbitrary choice of C. This means that proving two functions are equal requires us to consider the case in which a context passes them about, applies them to a few hundred arguments, and then ignores all those results and returns true. This roundabout usage of expressions we are interested in requires more sophisticated proofs.

Most proofs of contextual equivalence make use of a coinductive characterization of it. Contextual equivalence is the coarsest consistent congruence relation on terms. A consistent relation is one where if two terms are related by the relation than they coterminate. A congruence relation is an equivalence relation (reflexive, symmetric, transitive) which respects the structure of the terms.

To make this precise, a pair of relations $(\mathcal{R}, \mathcal{S})$ (the first on expressions, the second on commands) is said to respect the structure of terms if the following inferences are valid.

$$\frac{e_1 \mathrel{\mathcal{R}} e_2}{\lambda x{:}\tau.\, e_1 \mathrel{\mathcal{R}} \lambda x{:}\tau.\, e_2} \qquad \frac{e_1 \mathrel{\mathcal{R}} e_1' \qquad e_2 \mathrel{\mathcal{R}} e_2'}{e_1\ e_2 \mathrel{\mathcal{R}} e_1'\ e_2'} \qquad \frac{e_1 \mathrel{\mathcal{R}} e_2}{\Lambda\alpha.\, e_1 \mathrel{\mathcal{R}} \Lambda\alpha.\, e_2}$$

$$\frac{e_1 \mathrel{\mathcal{R}} e_2}{e_1[c] \mathrel{\mathcal{R}} e_2[c]} \qquad \frac{m_1 \mathrel{\mathcal{S}} m_2}{\mathsf{cmd}(m_1) \mathrel{\mathcal{R}} \mathsf{cmd}(m_2)} \qquad \frac{e_1 \mathrel{\mathcal{R}} e_2}{\mathsf{ret}(e_1) \mathrel{\mathcal{S}} \mathsf{ret}(e_2)}$$

$$\frac{e_1 \mathrel{\mathcal{R}} e_2 \qquad m_1 \mathrel{\mathcal{S}} m_2}{\mathsf{bnd}\ x \leftarrow e_1;\ m_1 \mathrel{\mathcal{S}} \mathsf{bnd}\ x \leftarrow e_2;\ m_2} \qquad \frac{e_1 \mathrel{\mathcal{R}} e_2 \qquad m_1 \mathrel{\mathcal{S}} m_2}{\mathsf{dcl}\ \alpha := e_1\ \mathsf{in}\ m_1 \mathrel{\mathcal{S}} \mathsf{dcl}\ \alpha := e_2\ \mathsf{in}\ m_2}$$

$$\frac{e_1 \mathrel{\mathcal{R}} e_2}{\mathsf{set}[\alpha](e_1) \mathrel{\mathcal{S}} \mathsf{set}[\alpha](e_2)} \qquad \frac{}{\mathsf{get}[\alpha] \mathrel{\mathcal{S}} \mathsf{get}[\alpha]}$$

The standard result is then captured by the following theorem.

Theorem 2.4. *Contextual equivalence on expressions and commands is the coarsest consistent congruence.*

We now turn to discussing logical relations for this language.

3 A Step-Indexed Logical Relation

Before diving into the various approaches for constructing a logical relation without step-indexing, it is well worth the time to see how a logical relation can be done with it. The purpose of this section is to sketch the complications intrinsic to any logical relation for state and show how step-indexing obliterates them, though at a high cost.

Our first step is to begin with a mapping from types to semantic types (merely sets of terms). In order to handle impredicative polymorphism we use Girard's method [30, 31]. See Harper [32] for an introduction of the technique. This means that our logical relation is of the form

$$[\![-]\!]_{-} : \mathrm{Type} \to \mathrm{TypeEnv} \to \mathcal{P}(\mathrm{Term} \times \mathrm{Term})$$

The central challenge is the meaning of $[\![\mathsf{cmd}(\tau)]\!]$: the action of the logical relation at commands. At an intuitive level, for two commands are rather like (partial) functions: they map heaps to heaps and a return value. Drawing inspiration from how logical relations for functions are defined, we might write the following for the definition of the logical relation at $\mathsf{cmd}(\tau)$:

$$\begin{aligned}
[\![\mathsf{cmd}(\tau)]\!]_{\eta} \triangleq \{(e_1, e_2) \mid \exists m_1, m_2.\ & e_i \mapsto^* \mathsf{cmd}(m_i) \wedge \\
& \forall h_1 \sim h_2.\ (m_1, h_1) \simeq (m_2, h_2) \wedge \\
& \quad \forall v_1, h_1', v_2, h_2'.\ ((m_i, h_i) \mapsto^* (\mathsf{ret}(v_i), h_i') \\
& \qquad \implies (h_1' \sim h_2' \wedge (v_1, v_2) \in [\![\tau]\!]_{\eta})
\end{aligned}$$

This definition states that two programs are equal as commands if they satisfy the following constraints:

1. They both run two commands.
2. When those commands are run on related heaps they terminate.
3. The resulting heaps and return values from this execution are related.

Here left undefined is the definition of when two heaps are related, $\sim$. This is in fact a major issue because there appears to be no good way to identify when two heaps ought to be equal. The first issue here is that semantic equality of terms (be it contextual or logical) is type-indexed. This means that in order to compare heaps pointwise for equality (a reasonable first approach) requires that we at least know the types of the entries. Furthermore, we should not compare these heaps for equality at all locations necessarily. Two heaps should only need to agree on the cells that the programs are going to use.

These two caveats are important if we want to prove programs to be equivalent which do not use the heap identically. For instance, consider the two programs:

$$\mathsf{dcl}\ \alpha := 1\ \mathsf{in}\ \mathsf{ret}(\mathsf{cmd}(\mathsf{get}[\alpha])) \qquad\qquad \mathsf{ret}(\mathsf{cmd}(\mathsf{ret}(1)))$$

These are contextually equivalent (because the assignable of the first program is hidden from external manipulation) and yet they allocate in different ways. So $\sim$ must not be *merely* pointwise equality of all cells in the most general case. It is also easy to see that proving that the programs above are equal requires showing that $h_1 \sim h_2$ if and only if $h_1(\alpha) = 1$. That is, this program doesn't merely require that heap cells contain values of some syntactic type, but they may need to belong to an arbitrary semantic type.

In order to reconcile these constraints, one thing is clear: the logical relation must somehow vary depending on the state that the heap is supposed to be in. It is simply not the case that programs that are equivalent in a heap where no cells are required to exist if and only if they're equivalent in a heap where one cell is required to exist.

The solution to this is called a Kripke logical relation. Kripke logical relations, and more generally sheaf models, are a recurring phenomenon in computer science [4, 6, 39, 45]. The underlying idea of Kripke logical relations, or sheaf models, is to abandon the notion of a single global truth and judge truth relative to a current state of the world. The notion of "state of the world" varies from application to application but is always a described as a partially ordered set, World. Elements of World represent the different states of knowledge we may have about the world and the partial order. The partial order, $w_1 \sqsubseteq w_2$, describes when w_2 is a consistent expansion of the knowledge of w_1.

For a Kripke logical relation, rather than considering $[\![\tau]\!]_\eta$, we work relative to a world w: $(e_1, e_2) \in [\![\tau]\!]_\eta(w)$. Importantly, Kripke logical relations should be monotone in relation to $\sqsubseteq$:

$$\forall w_1, w_2.\ w_1 \sqsubseteq w_2 \wedge (e_1, e_2) \in [\![\tau]\!]_{w_1} \implies (e_1, e_2) \in [\![\tau]\!]_{w_2}$$

Intuitively, if we know that some fact holds at a world, w_1, and we add more knowledge to w_1 to reach $w_2 \sqsupseteq w_1$ it should not reduce what we know to be true. Put in a more philosophical way: the absence of knowledge does not imply its negation.

In this case, the choice of Kripke world is meant to express the current state of the heap that programs are being compared at, or at least, what is known about it. What is this world concretely however? As a first cut, one could consider a simple collection of symbols and types. That is, World $\triangleq$ $\mathcal{P}_{\mathrm{fin}}(\text{Assignable} \times \text{Type})$ equipped with $\subseteq$. We can change our clause for the logical relation to take these worlds into account.

$$\begin{aligned}[\![\mathsf{cmd}(\tau)]\!]_\eta(w_1) \triangleq \{(e_1, e_2) \mid{} & \exists m_1, m_2.\ e_i \mapsto^* \mathsf{cmd}(m_i) \wedge \\ & \forall w_2 \sqsupseteq w_1.\ \forall h_1 {\sim_{w_2}} h_2.\ (m_1, h_1) \simeq (m_2, h_2) \wedge \\ & \quad \forall v_1, h_1', v_2, h_2'.\ ((m_i, h_i) \mapsto^* (\mathsf{ret}(v_i), h_i') \\ & \qquad \implies (\exists w_3 \sqsupseteq w_2.\ h_1' {\sim_{w_3}} h_2' \wedge (v_1, v_2) \in [\![\tau]\!]_\eta(w_3))\end{aligned}$$

These changes are largely forced. We must quantify over all possible w_2 extending w_1 at the beginning: if this was elided then $[\![\mathsf{cmd}(\tau)]\!]_\eta$ would not be monotone. The extension at the end, $w_3 \sqsupseteq w_2$, is so that the world may be updated to reflect the changes that were caused by allocating new cells or updating existing ones. Still unexplained is $h_1 \sim_w h_2$. At this point it can be defined in a slightly more refined way since w at least specifies what cells we ought to compare for equality. However, the notion of equality that we want is problematic. The problem appears in the *fundamental theorem*, the proof that the logical relation is reflexive.

- If some stronger notion of equality than logical equivalence, such as α-equivalence, is used the fundamental theorem will fail in the clause for $\mathsf{set}[\alpha](e)$.
- If a weaker equality than logical equivalence is used then the fundamental theorem will fail in the clause for $\mathsf{get}[\alpha]$.
- If logical equivalence itself is used, the definition will become ill-founded. This is because the heap may contain cells with a type larger than $\mathsf{cmd}(\tau)$.

What is needed is an maneuver in the vein of Girard's method. Instead of attempting to decide what the equality for a particular heap location should be in the definition of $\sim_w$, it should be told to us already by w. This idea, originating with the very early work on denotational models of state, means that our Kripke worlds should instead satisfy the relation:

$$\text{World} = \text{Assignable} \rightharpoonup \mathcal{P}(\text{Term} \times \text{Term})$$

Now the world extension relation is defined by the following.

$$w_1 \sqsubseteq w_2 \triangleq \text{Dom}(w_1) \subseteq \text{Dom}(w_2) \wedge \forall \alpha.\ w_1(\alpha) = w_2(\alpha)$$

This version is much more plausible. With the definition of the clause of the logical relation described previously together with the following definition of $\sim_w$ the logical relation is well-defined.

$$h_1 \sim_w h_2 \triangleq \forall \alpha \in \text{Dom}(w).\ (h_1(\alpha), h_2(\alpha)) \in w(\alpha)$$

The issue here is more subtle and causes the fundamental theorem to fail in the rule for allocation: what relation should we pick when a fresh cell is allocated? It seems that the only choice when allocating a cell of type τ at world w is to extend our world with $\alpha \mapsto [\![\tau]\!]_\eta(w)$. The complication arises when we allocate more cells later and move to a fresh world. The relation at α is now stale. It refers to an outdated world and doesn't allow for equivalences that are true at this new world but were previously false. To concretely see this, consider the program:

$$\begin{array}{l} \mathsf{cmd}(\mathsf{dcl}\ \alpha := \lambda x{:}\mathbb{N}.\,\mathsf{cmd}(\mathsf{ret}(1))\ \mathsf{in} \\ \qquad \mathsf{ret}(\mathsf{cmd}(\mathsf{set}[\alpha](\lambda x{:}\mathbb{N}.\,\mathsf{bnd}\ x \leftarrow \mathsf{cmd}(\mathsf{get}[\alpha]); \\ \qquad\qquad\qquad \mathsf{ret}(x+1))))) \end{array}$$

This style of program is used to encode recursion in this language. In this case, however, the central point of interest is that α is updated to contain a command which mentions α. In logical relation, α could only ever contain terms in $[\![\mathbb{N} \to \mathsf{cmd}(\mathbb{N})]\!](\emptyset)$. In particular, this never includes a command which mentions α. This prevents this perfectly type-safe program from being included in the logical relation and so the fundamental theorem must fail.

What is to be done here? The root of the issue is that when we allocate a cell it is impossible to determine precisely what programs will occupy it because programs in the cell may mention cells that are yet to be allocated at the time of the construction. What is needed is for the semantic type stored in a heap cell to vary according to the world. This fixes an asymmetry between the Kripke world and the logical relation: the Kripke world supposedly maps locations to semantic types but the semantic types (as determined by the logical relation) vary in the Kripke worlds. This leads us to the final form of the definition of Kripke worlds.

$$\text{World} = \text{Assignable} \rightharpoonup (\text{World} \to \mathcal{P}(\text{Term} \times \text{Term}))$$

Herein lies the rub: this definition of the set of worlds is precisely what is required for this logical relation but it is not a set. A simple cardinality argument shows that there can be no such set since it would have to be larger than its own power set. This is not an easily avoided problem. It is unknown how to simply avoid this using ingenuity in the choice of the Kripke world.

This is where step-indexing enters the picture: to break these circularities and solve the recursive equation up to an approximation.

With step-indexing, the space of semantic types becomes indexed by natural numbers and we will likewise index the world by these same natural numbers. The idea is that a world at stage n maps to assignables to semantic types which only vary at the previous $n-1$ stages, past $n-1$ they are simply constant. The issue is that this approach requires a number of complex definitions which obscure the underlying intent: to solve this recursive equation.

Instead of slogging through the classical step-indexed definitions, we will make use of a more modern categorical approach. Instead of working with sets, from the beginning we will work with sets varying over natural numbers: presheaves over ω (the preorder category given by $\mathbb{N}$ and $\leq$).

Definition 3.1. *A presheaf over a category $\mathbb{C}$ is a functor from $\mathbb{C}^{\mathsf{op}}$ to* **Set**. *Presheaves form a category with morphisms being natural transformations. This category is written $\widehat{\mathbb{C}}$.*

Example 3.2. *A presheaf over ω is a family of sets $(X_i)_{i \in \mathbb{N}}$ with a map $r_n : X_{n+1} \to X_n$ for all n called restriction maps. A map between presheaves is then a family of maps $f_n : X_n \to Y_n$ so that the following commutes for all n.*

$$\begin{tikzcd} X_{n+1} \arrow[r, "f_{n+1}"] \arrow[d, "r_n"'] & Y_{n+1} \arrow[d, "r_n"] \\ X_n \arrow[r, "f_n"'] & Y_n \end{tikzcd}$$

Rather than working directly with sets, we can instead work with presheaves synthetically. For instance, we can define the exponential of two presheaves rather than mucking about to create an implication which interacts properly with the step.

Lemma 3.3. *For all small diagrams of presheaves the (co)limit exists and is determined pointwise.*

The above theorem tells us, for instance, that if we want to form the product of two presheaves, X and Y, over ω, at time n it's just the product of the $X(n)$ and $Y(n)$. More generally, this lemma gives a wide-variety of ways to construct complex presheaves from simple ones without ever having to deal with the step manually. The main missing element is the ability to form exponentials, the categorical analog of function types. For presheaves these are slightly more complicated since functions between presheaves must respect the indexing structure.

Lemma 3.4. *Given two presheaves $X, Y : \widehat{\mathbb{C}}$, the exponential between them is*

$$(Y^X)(c) \triangleq \hom(\mathsf{y}(c) \times X, Y)$$

where y is the Yoneda embedding, defined by the following.

$$\mathsf{y}(c) = hom_{\mathbb{C}}(-, c)$$

This definition may seem abstract but it is, importantly, monotone and so determines a valid presheaf. The real power of this approach is that *it does not matter* that this definition is complex. The point is that this definition determines a function of presheaves in that it contains exactly and only elements we can apply. Beyond these two facts its construction is entirely irrelevant.

The constructions so far have given us a wide variety of constructible presheaves but nothing thus far has increased our expressive power of what we had in **Set**. For that we need the ability to solve certain recursive equations. In fact, it will turn out that we can solve *guarded* domain equations in $\widehat{\omega}$. In order to see this, first let us define a functor on $\widehat{\omega}$.

Definition 3.5. *The later functor, $\blacktriangleright : \widehat{\omega} \to \widehat{\omega}$, is defined on objects as follows.*

$$(\blacktriangleright X)(n+1) = X(n)$$
$$(\blacktriangleright X)(0) = \{\star\}$$

This family of sets is a presheaf in the obvious way: $r_0 = \lambda x.\, \star$ in $\blacktriangleright X$ and r_{n+1} of $\blacktriangleright X$ is the r_n of X. The action on morphisms is as follows.

$$(\blacktriangleright f)(n+1) = f_n$$
$$(\blacktriangleright f)(0) = \lambda x.\, \star$$

The later functor comes equipped with a natural transformation $\mathsf{next} : 1 \to \blacktriangleright$. Explicitly, this means next is a family of maps $\mathsf{next}_X : X \to \blacktriangleright X$ so that $\mathsf{next}_Y \circ f = f \circ \mathsf{next}_X$ for any $f : X \to Y$.

The later modality is the logical essence of step-indexing. The key insight of step-indexing is actually two-fold:

1. In order to show that two programs are indistinguishable, it suffices to show that they're indistinguishable for n steps for any n.
2. When showing that two programs are indistinguishable from other programs at n steps it suffices to show that subcomponents are only indistinguishable for $n-1$ steps.

If this second point was not the case, then it would never be possible to decrement the step-index and the exercise would have been largely moot. The later modality internalizes the idea of "$n-1$ steps" without ever mentioning n. Using $\blacktriangleright X$ we can talk about (uniformly) constructing an element at stage n from an element at stage $n-1$. The next operation even crystallizes the monotonicity of the logical relation: if we have an element at stage n we can always construct one at stage $n-1$.

A crucial point about using natural numbers for step-indexing is that natural numbers are well-founded; there's no way to pick a natural number that we can decrement forever. This justifies reasoning by induction on the natural numbers and, in particular constructing an object at any stage provided we can take a construction of it at stage $n-1$ and construct it at stage n. This principle may be expressed with the later modality: it's precisely the existence of a family of morphisms:

$$\mathsf{fix}_A : A^{\blacktriangleright A} \to A$$

These functions express Löb induction: in order to construct A, it suffices to assume $\blacktriangleright A$ and construct A.

These morphisms allow us to take a particular sort of fixed-point at any level of maps $\blacktriangleright A \to A$. These morphisms belong to the class of contractive morphisms: morphisms $X \to Y$ that can be factored as $X \to \blacktriangleright X \to Y$. In particular, suppose we have some $f : B \times \blacktriangleright A \to A$. Then the following equality holds:

$$\mathsf{fix}_A \circ \widehat{f} = f \circ \left\langle 1, \mathsf{next}_A \circ \mathsf{fix}_A \circ \widehat{f} \right\rangle$$

Written out using more type-theoretic notation this is easier to process.

$$\forall a, b.\ \mathsf{fix}_A(f(b, -))(a) = f(b, \mathsf{next}_A(\mathsf{fix}_A(f(b))(a)))$$

This fixed-point construction allows us to build many interesting structures inside this category. Through the inclusion of universes [35], one can even construct solutions to certain (small) domain equations entirely internally to the category [14]. The basic idea is that $\widehat{\omega}$ can be easily made to support an object, $\mathcal{U}$, for which global sections (maps $1 \to \mathcal{U}$) classify other smaller objects. This is the categorical analog of a Grothendieck universe [51] and proves to be an important concept in the semantics of dependent type theory. Then, by using $\mathsf{fix}_{\mathcal{U}}$ it is possible to construct the recursively defined presheaves necessary to build logical relations. This approach, while not always explicit, is latent in the work done in constructing logical relations in guarded type theory and guarded logics [24, 36, 41].

We will consider the more traditional approach of using proper domain equations rather than just small ones. With this methodology, we model equations as functors and solutions as fixed points of these functorsf. This approach in domain theory originates from Smyth and Plotkin [50] and subsumed prior work by Scott [49] constructing recursive domain equations by hand.

One potential issue is that we would like to solve domain equations which are not strictly speaking functorial. In particular, we want to handle the case where the equation contains both positive and negative occurences. We handle these by viewing them as functors $F : \widehat{\omega}^{\mathsf{op}} \times \widehat{\omega} \to \widehat{\omega}$ in the style of Pitts [42] for instance. The question then becomes, for what functors can we construct an object, I, so that $F(I, I) \cong I$, an invariant object. This construction is a generalization of the theorem of America and Rutten [5]. The fully general statement the theorem can be found in Birkedal et al. [16, 17]. For our purposes though a much less general theorem suffices.

Definition 3.6. *A functor, F, is said to be locally contractive if there is a family of contractive morphisms*

$$f_{X_1,X_2,Y_1,Y_2} : X_1^{X_2} \times Y_2^{Y_1} \to F(X_2, Y_2)^{F(X_1,Y_1)}$$

So that this family respects composition and identity and so that for each $\langle g, h\rangle : 1 \to X_1^{X_2} \times Y_2^{Y_1}$ then the following equation holds.

$$f_{X_1,X_2,Y_1,Y_2} \circ \langle g, h\rangle = \widehat{F(\widehat{g}, \widehat{h})}$$

If just the latter condition holds (no respect for identity or composition) F is said to have a strength f and if the family of morphisms is not contractive we shall call the functor locally nonexpansive.

This definition is subtly different than some presentations of this theorem, which require only that F have a strength comprised of contractive morphisms. This version essentially states that F must be an enriched functor for the category $\widehat{\omega}$ enriched over itself. The central theorem, Theorem 3.9 is true even in this weakened version but I am unable to find a proof of this fact. The relative strengthening that this generality provides seems barren as well; none of the examples of the fixed point theorem used either in this section or Section 6 require it.

Example 3.7. *The functors $\times$, $\to$, and $+$ are all locally nonexpansive. The functor $\blacktriangleright$ is locally contractive. Additionally, any locally contractive functor is locally nonexpansive.*

Lemma 3.8. *The composition of locally nonexpansive functors is locally nonexpansive. The composition of a locally contractive functor with a locally nonexpansive functor is locally contractive.*

We may now state the theorem which makes working a $\widehat{\omega}$ so appealing.

Theorem 3.9. *For any locally contractive functor F there exists an object I, unique up to isomorphism, so that $F(I, I) \cong I$.*

This theorem allows us to solve a wide variety of domain equations, including a small modification of the crucial one for Kripke worlds. First, let us define $\mathsf{UPred}(S)$ as a "time-varying predicate" on a set S. This will be the analog of the sets of terms used in step-indexed models.

$$\mathsf{UPred}(X)(n) \triangleq \{(m, x) \mid x \in X \land m < n\}$$

There is an evident restriction mapping sending a uniform predicate at stage $n+1$ to the uniform predicate containing the entries indexed by numbers smaller than n. Next, the presheaf of partial maps from assignables to presheaves can be defined as follows:

$$(\text{Assignable} \rightharpoonup X)(n) \triangleq \{f \in X(n)^F \mid F \subseteq \text{Assignable} \land F \ \mathsf{finite}\}$$

Importantly, it is not hard to see that Assignable $\rightharpoonup -$ defines a functor which is locally nonexpansive. The final piece necessary is a subobject of the exponential $Y^{\text{Assignable} \rightharpoonup X}$ which isolates those functions which are monotone. In order to define this, we must assume that Y comes equipped with a partial order. Viewed from the external perspective this is a family of relations indexed by natural numbers which respects reindexing. If one views the partial order internally, however, it is just a normal partial order presented categorically. The subobject desired can be defined internally in a way which is obviously correct using the internal logic of $\widehat{\omega}$ as follows.

$$\begin{aligned}&(\text{Assignable} \rightharpoonup X) \xrightarrow{\mathsf{mon}} Y \triangleq \\ &\qquad \{f : Y^{\text{Assignable} \rightharpoonup X} \mid \forall w_1 w_2 : \text{Assignable} \rightharpoonup X.\ w_1 \sqsubseteq w_2 \implies f(w_1) \leq f(w_2)\}\end{aligned}$$

Constructed externally this presheaf is somewhat more difficult to construct, but still quite possible. It's

$$\begin{aligned}&((\text{Assignable} \rightharpoonup X) \xrightarrow{\mathsf{mon}} Y)(n) = \\ &\qquad \{f : (Y^{\text{Assignable} \rightharpoonup X})(n) \mid \forall m \leq n.\ \forall w_1, w_2 : (\text{Assignable} \rightharpoonup X)(n). \\ &\qquad\qquad w_1 \sqsubseteq_n w_2 \implies f(m)(\star, w_1) \leq_n f(m)(\star, w_2)\}\end{aligned}$$

This external definition is neither more precise nor more intuitive, so it is more helpful to read the internal version and understand that all the quantifiers in the predicate implicitly handle the steps correctly. A more technical discussion of the internal language of $\widehat{\omega}$ is postponed until the end of this section.

We may define the following contractive functor.

$$X \mapsto \blacktriangleright((\text{Assignable} \rightharpoonup X) \xrightarrow{\mathsf{mon}} \mathsf{UPred}(\text{Term} \times \text{Term}))$$

Solving this for a fixed point gives the desired presheaf of Kripke worlds. Defining the rest of the logical relation proves to be quite straightforward now that the right definition of Kripke worlds is in place. The type of our logical relation is slightly different now; it's defined as a monotone map:

$$[\![-]\!]_{-} : \text{Type} \to \text{TypeEnv} \to \text{World} \xrightarrow{\textsf{mon}} \textsf{UPred}(\text{Term} \times \text{Term})$$

Here $\xrightarrow{\textsf{mon}}$ is an subobject of the exponential in the presheaf category so it is indexed by natural numbers and subject to a naturality condition. Then the crucial clause of our logical relation can be defined as follows.

$$\begin{aligned}
[\![\textsf{cmd}(\tau)]\!]_\eta(n)(w_1) \triangleq \{(k, e_1, e_2) \mid {} & k_1 \leq n \wedge \exists m_1, m_2.\ e_i \mapsto^* \textsf{cmd}(m_i) \wedge \\
& \forall w_2 \sqsupseteq_{k_1} r^n_{k_1}(w_1).\ \forall h_1 \sim^{k_1}_{w_2} h_2.\ (m_1, h_1) \simeq (m_2, h_2) \wedge \\
& \forall d.\ \forall v_1, h_1', v_2, h_2'.\ ((m_1, h_1) \mapsto^d (\textsf{ret}(v_1), h_1') \wedge (m_2, h_2) \mapsto^* (\textsf{ret}(v_2), h_2')) \\
& \implies (\exists w_3 \sqsupseteq r^{k_2 - d} w_2.\ h_1' \sim_{w_3} h_2' \wedge (v_1, v_2) \in [\![\tau]\!]_\eta(k_2 - d)(w_3))
\end{aligned}$$

We can now also define $\sim$ making use of the following isomorphism.

$$\iota : \text{World} \cong \blacktriangleright(\text{Assignable} \rightharpoonup \text{World} \xrightarrow{\textsf{mon}} \textsf{UPred}(\text{Term} \times \text{Term}))$$

The following definition is then quite naturally expressed internally. The external version merely involves a preponderance of indices. Let us suppose that w is a world at step n, then:

$$h_1 \sim_w h_2 \triangleq$$
$$\begin{cases} \forall \alpha \in \text{Dom}(\iota(w)).\ (n-1, h_1(\alpha), h_2(\alpha)) \in \iota(w)(\alpha)(r^{n-1}(w)) & n > 1 \\ \text{true} & \text{otherwise} \end{cases}$$

The essence of the definition is the self-application. A heap is related to another heap at a world if all of the points are related relative to the same world. It was this circularity and self-application which drove us to step-indexing in the first place.

The rest of the logical relation is entirely standard step-indexing and the curious reader is referred to Ahmed [4].

A final consideration here is that the version of step-indexing presented in this paper is "half-baked". We have eschewed explicit indexing in the entire construction of the semantic domains and made use of ▶ and presheaves, but then chosen to define the logical relation with explicit indexing. All the same issues that applied to the construction of the domain of discourse for the logical relation apply the actual definition of the logical relations itself. Since our definition of the logical relation is just constructing a subpresheaf pointwise using standard logical formulas and passing around the index, can we isolate the handling of the index and abstract away from it?

A powerful approach to handling this comes from topos theory. Topos theory is a subfield of category theory concerned with studying categories which are similar to **Set**. It is a standard result in topos theory that all presheaf categories are toposes and so all the tools of topos theory are applicable to $\widehat{\omega}$. For instance, the explicit definition of $\textsf{UPred}(A)$ can be replaced with the topos-theoretic definition of power sets. Of particular interest is the "internal logic"[4] of the topos. It turns out that $\widehat{\omega}$ supports a version of intuitionistic higher-order logic whose interpretation is precisely the standard set-theoretic interpretation with a little extra work to pass around the index. One can take a presheaf, A, and a predicate, ϕ, on it and form a subpresheaf $\{a \in A \mid \phi(a)\}$ so that every element of this comprehension satisfies ϕ as one would hope.

The internal logic of $\widehat{\omega}$ includes everything we need to define our logic including even a version of Löb induction and a logical version of later, $\triangleright$. Working this way sweeps the step-index entirely under the rug and obliterates the requirement to perform explicit index math anywhere. Instead, the frustrating mismatch of having a fact at $n-1$ and needing it at n manifests through having $\triangleright\phi$ and needing ϕ. Proper index discipline is replaced by the proper handling of the later modality. For instance, here is the definition of world satisfaction working in the internal logic.

$$h_1 \sim_w h_2 \triangleq \forall \alpha \in \mathrm{Dom}(w).\ \mathsf{succ}(\iota(w) \circledast \mathsf{next}(w) \circledast (h_1, h_2))$$

No explicit steps are present only a handful of logical operations to handle the new type-constructor later. This approach to step-indexing, whether explicitly topos-theoretic or not is the subject of a long line of work. Importantly, this logical approach to step-indexing enables abstraction from the natural numbers themselves and suggests more general notions of step-indexing [52]. The most approachable introduction to this approach is Dreyer et al. [24] and Birkedal et al. [17] provides a thorough topos-theoretic grounding explicitly using Kripke-Joyal semantics.

4 Tying the Knot Using Domains

This section describes the first and most obvious approach to building a step-index-free logical relation for state. It would seem that the real crux of the issue was this Kripke world. Nothing past the construction of the Kripke world in Section 3 was particularly challenging, especially when a logical approach is taken. So a natural approach is to find a different setting in which to solve the following equation

$$\mathrm{World} \cong \mathrm{Assignable} \rightharpoonup (\mathrm{World} \xrightarrow{\mathsf{mon}} \mathcal{P}(\mathrm{Term} \times \mathrm{Term}))$$

Traditionally, domains have been used for exactly this purpose. The impetus of domain theory as an investigation separate from order theory in general was the observation by Dana Scott that domains possessed solutions to equations that sets did not. For instance, a model of the lambda calculus may be constructed by solving the equation

$$D \cong D \to D$$

[4]In fact, the internal logic of a topos can be generalized to a full dependent type theory with a distinguished impredicative universe. This type theory provides even more flexibility than the logic. In it, we could have defined the entirety of our construction of Kripke worlds in a setting analogous to programming in type theory.

This equation is not possible to solve in sets without trivializing D. In fact, Theorem 3.9 uses a construction derived originally from domain theory itself.

4.1 Domain Theory, Briefly

A domain is a particular sort of partial order which comes equipped with enough structure to accurately describe a notion of continuous functions upon it.

Definition 4.1. *In a partial order P, a directed set $D \supseteq P$ is a set so that $D \neq \emptyset$ and if $a, b \in D$ then there is a $c \in D$ so that $a, b \leq c$.*

Definition 4.2. *A domain*[5] *is partial order possessing joins (least upper bounds) of all directed subsets. Given a directed subset D we write this join as $\bigvee D$.*

Intuitively, elements of a domain represent a (partial) piece of knowledge. Moving to a larger element represents moving to a piece of knowledge that extends what was already known. The ability to construct joins of directed sets fits into this intuition as the ability to take a great many fragments of compatible knowledge and glue them together.

This intuition primarily comes from the example domain of a partial function. The function is not everywhere defined so it is not a complete piece of knowledge, but given a collection of partial functions which agree on the intersection, they can be glued together. More precisely, fix our underlying set of $S \rightharpoonup D$ to be partial functions from some set S to a domain D. The ordering on these functions is given by the following:

$$f_1 \sqsubseteq f_2 \triangleq \forall s \in S.\ f_1(s) \Downarrow \implies f_1(s) = f_2(s)$$

Then, joins are constructed on directed sets as follows:

$$(\bigvee_i f_i)(s) = \bigvee \{d \mid \exists i.\ f_i(s) = d\}$$

Showing that this is a partial function relies crucially on the directedness of $(f_i)_i$.

The extra structure on domains gives rise to a new class of maps that preserve this structure. These maps are called (Scott) continuous in that they preserve joins.

Definition 4.3 (Scott continuity)**.** *A map between domains is (Scott) continuous if it preserves the joins of directed sets. That is, if $f : D \to E$ and $I \subseteq D$ is directed, then $f(\bigvee I) = \bigvee\{f(i) \mid i \in I\}$.*

Calling this definition continuity can be justified by considering the Scott topology on domains, in which continuous functions are precisely those which preserve upper bounds. For a complete account of the theory of domains, see for instance Abramsky and Jung [2], Gierz et al. [29].

[5]In this text domains are directed complete partial orders as opposed to any of the myriad varierties of domains one might have chosen.

Theorem 4.4. *Domains form an a category,* **Dcpo***, where morphisms are continuous functions. This category is bicartesian closed, in particular the set of continuous maps between two domains is a domain.*

The interesting portion of domain theory for these purposes is the ability to solve domain equations. Specifically, locally continuous functors of domains again have solutions.

Definition 4.5. *A functor, $F : \mathbf{Dcpo} \to \mathbf{Dcpo}$ is locally continuous if its action on morphisms is a continuous function. That is, there is a continuous map of domains $B^A \to F(B)^{F(A)}$. This definition generalizes readily to a mixed variance n-ary functor.*

Theorem 4.6. *Any mixed variance locally continuous functor $F : \mathbf{Dcpo}^{\mathsf{op}} \times \mathbf{Dcpo} \to \mathbf{Dcpo}$ has an invariant object I so that $F(I, I) \cong I$.*

4.2 Domains for State

The domains involved in expression the desired equation are quite straightforward. Power sets are always a complete lattice and therefore a domain. Scott continuous functions are always monotone so instead of considering monotone functions it's more natural to pick Scott continuous functions to get a domain. Finally, partial maps form a domain as shown above with the small complication that they cannot be limited to finite partial maps.

This gives us a proper functor $F : \mathbf{Dcpo}^{\mathsf{op}} \to \mathbf{Dcpo}$ defined by

$$\begin{aligned} D &\mapsto \text{Assignable} \rightharpoonup (D \xrightarrow{\mathsf{mon}} \mathcal{P}(\text{Term} \times \text{Term})) \\ f &\mapsto \lambda x.\ \lambda\alpha.\ \lambda d.\ x(\alpha)(f(d)) \end{aligned}$$

There is only one exceptional feature of all of this, the ordering on the partial map aspect of this domain is nonstandard. A Kripke world is supposed to map an assignable to a semantic type. The ordering on these worlds should let us add new mappings, but old mappings should stay the same. They must in fact, if a location is allocated so that it can be either `true` or `false`, it would be a real problem if suddenly it could also be a natural number. This ordering, explicitly, is then defined by the following formula.

$$w_1 \sqsubseteq w_2 \iff \forall \alpha \in \text{Dom}(w_1).\ w_1(\alpha) \simeq w_2(\alpha)$$

This ordering, however natural, means that F is not locally continuous and therefore there is no way to construct a solution to it.

Supposing for a moment that a solution to F did exist. There would still seemingly be a problem because one of these worlds could have full support, making modeling allocation impossible. This can be avoided by defining the logical relation on compact elements [29] of the world domain, precisely those worlds enjoying only finite support.

By defining the logical relation on these worlds, it is trivial to extend it to an arbitrary world by taking $[\![\tau]\!](w)$ on such a world to be the join of the results of $\llbracket \tau \rrbracket$ on all the compact elements below w. This is automatically continuous and recovers exactly $\llbracket \tau \rrbracket$ on compact elements which are, after all, the worlds of interest.

5 Understanding State through (Syntactic) Minimal Invariance

Now that domains are seen to be unhelpful, there is some question about where to turn next. There are not many well-studied algebraically complete categories [28] which support interesting denotational semantics. The semantic structures for step-indexing that support solving recursive domain equation require more than continuity or approximation. They require a notion of closeness for two semantic types. In $\widehat{\omega}$ for instance, there was a natural notion of when two points $p, q : 1 \to P$ were close by checking at what stage they became equal, eg when $p_n = q_n : \{\star\} \to P$.

Crucially, in order for this to make sense for semantic types every equality was judged relative to a step. Two semantic types are equal at n if they agree on equalities that hold at stage n or earlier, eg, equalities which hold if the two sides are only allowed to run for n steps. Without step-indexing what could this be replaced by?

The answer is suggested by an idea in domain theory to resolve a seemingly unrelated problem. The observant reader may have noticed that there is a crucial strengthening that Theorem 3.9 provides over Theorem 4.6: the former provides unique solutions. In fact, there may be many nonisomorphic domains satisfying a given equation. This state of affairs can prove troublesome for semantics in domains because it means that not all the properties of the domain are determined by the equation. There may be "exotic" elements which are not required to satisfy the domain equation but are nevertheless present which impedes reasoning about recursive domain equations by induction or similar. Pitts [42] proposes a solution to this by imposing an additional requirement on the solution to a domain equation: minimal invariance.

Minimal invarianceis the statement that given a D so that $D \cong F(D, D)$ the following equation holds.

$$\bigvee_i \pi_i = 1$$

The family of functions π_i is determined by induction on i.

$$\begin{aligned} \pi_i &: D \to D \\ \pi_0 &\triangleq \bot \\ \pi_{i+1} &\triangleq F(\pi_i, \pi_i) \end{aligned}$$

The intuition behind this definition is that if π_i converges to 1 then every element in D must have arisen as the limit of elements computed by the inclusion $F^n(\bot, \bot) \to D$ which can be viewed as semantic requirement that elements of

D are finite[6]. It is not difficult to prove that if $D \cong F(D)$ and $E \cong F(E)$ with D and E both being minimally invariant then $E \cong D$ and the standard D_∞ construction produces a minimally invariant domain. Therefore, domain equations have a unique minimally invariant solution and these satisfy a form of induction.

An interesting observation made originally by Birkedal and Harper [12] and extend by Crary and Harper [22] is that for the class of domains arising naturally in semantics (so called "universal domains") the projection functions, π_i, are computable syntactically. Additionally, since contextual equivalence can be generalized to contextual approximation in a natural way, minimal invariance can be stated as a syntactic property of the programming language itself. In this setting, projections are naturally type-indexed and this is denoted $\pi^i_\tau : \tau \to \tau$. The definition of π^i_τ behaves almost as an η-expansion where after i η-expansions the function diverges.

Definition 5.1. *Two terms $\Delta; \Gamma \vdash_\Sigma e_1, e_2 : \tau$ are said to be contextually approximate at type τ, written $\Delta; \Gamma \vdash_\Sigma e_1 \lessapprox e_2 : \tau$ if for all contexts $C : (\Delta; \Gamma \vdash_\Sigma \tau) \rightsquigarrow (\cdot; \cdot \vdash_{\Sigma'} \mathsf{cmd}(\tau'))$, $C[e_1] \mapsto^* \mathsf{cmd}(m_1)$ and $C[e_2] \mapsto^* \mathsf{cmd}(m_2)$ such that for any heap $h : \Sigma$ then $(m_1, h) \Downarrow \implies (m_2, h) \Downarrow$.*

We will adopt the traditional abuse of abandoning the context annotations and just writing $e_1 \lessapprox e_2$ when the contexts and τ are obvious.

Definition 5.2. *A family of terms $(e_i)_{i \in I}$ is said to have a limit e if for all $i \in I$ $e_i \lessapprox e$ and e is the smallest element with this property.*

Since limits are only unique up to contextual equivalence, it is not necessarily well defined to write $\bigvee_i e_i$ as if it were a term rather than an equivalence class of terms. We will instead often make use of the notation $\bigvee_i e_i = e$ to signify that e belongs to this equivalence class. The syntactic version of minimal invariance can now be precisely stated.

Theorem 5.3. *For all $\Gamma \vdash \tau : \mathsf{T}$ and $e : \tau$, $\bigvee_i \pi^i_\tau \ e = e$.*

The idea behind using syntactic minimal invariance is that instead of defining a single logical relation $[\![-]\!]_-$ a pair of logical relations is defined instead: $\lceil - \rceil_-$ and $\lfloor - \rfloor_-$ (pronounced "definitely" and "possibly"). The definition of these two logical relations are mutually recursive and provide a decomposition of the logical relation two monotone halves. The idea is that one should think of $\lceil \tau \rceil_\eta$ containing equalities that are certainly true but it does not contain all of them. On the other hand $\lfloor \tau \rfloor_\eta$ contains all equalities that are true as well as a few that might not be. This suggests a natural relationship between these two sets where $\lceil \tau \rceil_\eta \subseteq \lfloor \tau \rfloor_\eta$. The crucial move that syntactic minimal invariance provides is to show the following.

$$\forall i.\ (e_1, e_2) \in \lceil \tau \rceil_\eta \implies (\pi^i_\tau\ e_1, \pi^i_\tau\ e_2) \in \lfloor \tau \rfloor_\eta$$

[6]Or rather, determined as the limits of "finite" elements. This is not to be confused here with *compact* elements and algebraic domains which share a similar intuition but require technically different properties since there is no reason to assume that F preserves algebraicness

Then, by showing the comparatively easy lemma that $\lfloor \tau \rfloor_\eta$ is always closed under limits when η is it immediately follows that $\lceil \tau \rceil_\eta = \lfloor \tau \rfloor_\eta$. This maneuver is useful because these two definitions are naturally monotone. It is then easy to express a natural definition of recursive types, as is done in Crary and Harper [22] for instance. The only complication is that environments must generalize from relations to birelations, a map to a pair of relations rather than a map to a single relation. Define η^{op} to be the environment mapping α to the relations $\eta(\alpha)$ with the components swapped. The crucial insight is that when the variance in the definition of the logical relation swaps, we switch from $\lceil - \rceil$ to $\lfloor - \rfloor$ and vice-versa, for instance:

$$\lceil \tau_1 \to \tau_2 \rceil_\eta = \{(e_1, e_2) \mid \forall (a_1, a_2) \in \lfloor \tau_1 \rfloor_\eta .\ (e_1\ a_1, e_2\ a_2) \in \lceil \tau_2 \rceil_\eta\}$$

This approach seems to naturally fit recursive types and one might wonder then if it can be scaled to support state just as step-indexing does.

The key insight is that while the distinction of $\lceil - \rceil / \lfloor - \rfloor$ is not useful, projection functions give us a notion akin to step-indexing. We can say that two semantic types are equal at stage i if they are equal upon truncating all the programs in the equalities by π^i. In order to define π^i_τ we need a divergence at the level of expressions, this can be easily accomplished by adding fixed points or just adding a formal term $\bot$ so that $\bot \mapsto \bot$.

$$\begin{aligned}
\pi^0_\tau &= \lambda x{:}\tau.\, \bot \\
\pi^{i+1}_{\tau_1 \to \tau_2} &= \lambda f{:}\tau_1 \to \tau_2.\, \lambda x{:}\tau_1.\, \pi^i_{\tau_2}\ (f\ (\pi^i_{\tau_1}\ x)) \\
\pi^{i+1}_{\forall \alpha.\, \tau} &= \lambda f{:}\forall \alpha.\, \tau.\, \Lambda \alpha.\, \pi^i_\tau\ (f[\alpha]) \\
\pi^{i+1}_{\mathsf{cmd}(\tau)} &= \lambda x{:}\mathsf{cmd}(\tau).\, \mathsf{cmd}(\mathsf{bnd}\ x' \leftarrow x;\ \mathsf{ret}(\pi^i_\tau\ x'))
\end{aligned}$$

This approach, while effective for recursive types, is not quite what is needed to handle state. After all, the issue in Section 3 was not the definition of $[\![\mathsf{cmd}(\tau)]\!]_\eta(w)$. The issue was was the construction of the set in which w lives. On the other hand, the projections themselves are interesting, they provide a way to construct an indexing of semantic types without actually counting steps. This approach was explored in Birkedal et al. [16] and successfully used to construct a logical relation for a similar language. This work, however, made use of the normal domain theoretic projections rather than their syntactic counterparts. The question of producing a logical relation for our language is then reduced to the question of producing a syntactic account of Birkedal et al. [16].

In this account, monadic computations are represented by store-passing.

$$\begin{aligned}
\text{State} &= \text{Assignable} \rightharpoonup D \\
D &\cong (D \to D) \oplus (D \times D) \oplus ... \oplus \text{State} \to (\text{State} \times D)
\end{aligned}$$

Projections, π^i, work by casing on which branch of the domain equation the argument is in and proceeding accordingly. This is analogous to how with syntactic minimal invariance the projection function is split according the type of the argument. Most of the clauses are standard with the interesting one being the last one for commands. Denote the injection of State $\to$ (State $\times D$) into D by $\mathsf{cmd}(-)$. Then π^{i+1} on this satisfies the following formula.

$$\pi^{i+1}(\mathsf{cmd}(f)) = \mathsf{cmd}(\lambda s.(\mathsf{trunc}^i(\pi_1 f(s)), \pi^i(\pi_2 f(s))))\mathsf{trunc}^i(s) = \pi^i \circ s$$

In words: truncation on a command runs the command a point-wise truncates the heap and then truncates the return value as well as the resulting heap.

In order to implement this syntactically, there's an issue. How can it be possible to replicate in a setting where accessing the whole heap is forbidden? In fact, this cannot be implemented as is. This presents a serious issue in scaling up syntactic minimal invariance and there is no clear solution.

If, for instance, a new command $\mathsf{trunc}_i \div \text{unit}$ is added to the language, it is possible to express the projections. With the other definition left unchanged the new case, $\pi^{i+1}_{\mathsf{cmd}(\tau)}$, is as follows.

$$\begin{aligned}\lambda c{:}\mathsf{cmd}(\tau).\, \mathsf{cmd}(&\mathsf{bnd}\ \_ \leftarrow \mathsf{cmd}(\mathsf{trunc}_i);\\ &\mathsf{bnd}\ x \leftarrow c;\\ &\mathsf{bnd}\ \_ \leftarrow \mathsf{cmd}(\mathsf{trunc}_i);\\ &\mathsf{ret}(\pi^i_\tau\ x))\end{aligned}$$

If this is added then Theorem 5.3 holds when we add the following rule for evaluating trunc^i.

$$\frac{}{(\mathsf{trunc}^0, h) \mapsto^* (\mathsf{trunc}^0, h)} \qquad \frac{h(\alpha) = \pi^i_\tau\ \alpha}{(\mathsf{trunc}^{i+1}, h) \mapsto^* (\mathsf{ret}(()), h')}$$

It is worth noting the strangeness of the last rule which morally "truncates the full heap". This strangeness is twofold. It bakes in the π_τ functions into the very operational semantics despite them just being chunks of user-defined code. It also requires that the heap be in some way typed so that when projecting at some particular cell it is possible to determine the type to truncate at.

Beyond the strangeness of the rule, however, it works. A logical relation can be defined using Kripke worlds which index semantic types by truncation. This is problematic, however, because it changes the notion of contextual equivalence. By adding new constructs to the language new contexts are added and this impacts contextual equivalence. In this extended language, programs that ought to be equal are no longer. For instance, the following program should be equivalent to $\mathsf{cmd}(\mathsf{cmd}(\mathsf{ret}(\lambda x{:}\tau.\, x)))$ but is not.

$$\begin{aligned}&\mathsf{cmd}(\mathsf{dcl}\ \alpha := \lambda x{:}\tau.\, x\ \mathsf{in}\\ &\quad \mathsf{ret}(\mathsf{cmd}(\mathsf{get}[\alpha])))\end{aligned}$$

The issue is that another part of the program could truncate the heap, replacing $\lambda x{:}\tau.\, x$ with a truncation of itself. This truncation would diverge strictly more than the original and thus the second piece of code only contextually approximates the first. This weakening is discussed more in subsequent work to

Birkedal et al. [16], for instance in Birkedal et al. [18]. In our case, however, this restriction is damaging. It makes it impossible to prove any information-hiding results using our logical relation and this makes it difficult to prove that certain effects are benign.

This puts a stop to naively extending syntactic minimal invariance to our language with state. Instead, a better approach seems to be to isolate a more type-theoretic characterization of the difficulty of a logical relation for state.

6 A Logical Relation in $\widehat{\omega}$ with Syntactic Minimal Invariance

The final proposed approach to handling logical relations for state is to instead study a language which allows us to merely encode state. This is not sufficient to solve many of the original goals of this work but it's a simpler problem and a natural stop along the way to a proper logical relation. The natural candidate for a logical relation is to consider (guarded) recursive kinds as described by Pottier [46]. The motivation for doing this is that a language with guarded recursive kinds is quite complex but in a very different way than a language with state. It does not require Kripke worlds in particular, the recursion is instead present in the kind structure. By being present in the structure of types itself it is easier to handle with type-based methods.

Recursive kinds require us to switch to a richer language where the types themselves support computation. For our purposes, the language is given by the following grammar.

$$\begin{array}{lcl} e & ::= & x \mid \lambda x{:}\tau.\, e \mid e\ e \mid \Lambda\alpha{:}\kappa.\, e \mid e[c] \\ c, \tau & ::= & \alpha \mid \tau \to \tau \mid \forall\alpha{:}\kappa.\, \tau \\ & \mid & \lambda\alpha{:}\kappa.\, c \mid c\ c \mid \mathsf{in}\, c \mid \mathsf{out}\, c \\ \kappa & ::= & j \mid \mathsf{T} \mid \kappa \to \kappa \mid \mu j.\, \kappa \end{array}$$

This language features a simple term language but a much richer type language, here called *constructors*. Constructors allow for computation which produces types. This is formally captured by two judgments: $\Omega; \Delta \vdash c : \kappa$ and $\Omega; \Delta \vdash c_1 \equiv c_2 : \kappa$. The first judgment specifies that a constructor c has a kind κ in a context of kind variables Ω and constructor variables Δ. The second specifies when two constructors are equal. This latter judgment is related to the actual expressions of the language by the following rule.

$$\frac{\Omega; \Delta; \Gamma \vdash e : \tau_1 \qquad \Omega; \Delta \vdash \tau_1 \equiv \tau_2 : \mathsf{T}}{\Omega; \Delta; \Gamma \vdash e : \tau_2}$$

The constructor language typing judgments are standard for Fω (see Barendregt [11] for an explanation) except for the following rules.

$$\frac{\Omega, j \vdash \kappa : \mathsf{kind}}{\Omega \vdash \mu j.\, \kappa : \mathsf{kind}} \qquad \frac{\Omega; \Delta \vdash c : \mu j.\, \kappa}{\Omega; \Delta \vdash \mathsf{out}\, c : [\mu j.\, \kappa / j]\kappa} \qquad \frac{\Omega; \Delta \vdash c : \mu j.\, \kappa}{\Omega; \Delta \vdash \mathsf{out}\, c : [\mu j.\, \kappa / j]\kappa}$$

The calculus described so far is a version of Fω with recursive kinds. This language is quite expressive because recursive kinds allow for recursive constructor definitions of arbitrary kind through encoding the Y-combinator at the constructor level. It is exceptionally poorly behaved because recursive kinds allow for constructors which do not have a weak head normal form. This forces choices in the language that seem to have no good answers. For instance, what terms belong to a constructor of kind T that has no weak head normal form.

There is a technique for potentially approaching a logical relation for even this language with general recursive types. It draws on two key ideas, one from Section 4 and one from Section 5. We define a pair of logical relations $\lceil - \rceil$ and $\lfloor - \rfloor$ as in Section 5. This logical relation is defined on the *normal forms* of constructors however. In order to handle the cases where a term has an infinite normal form or simply diverges, we actually work with finite fragments of the Böhm tree of a term [11]. Two terms are related at a type with an infinite normal form if and only if they are in all finite approximations of that type. In this case, there is actually a meaningful difference between $\lceil - \rceil$ and $\lfloor - \rfloor$: if we reach a node in the Böhm tree which indicates that more information exists but we are not allowed access to it in this approximation, say ?, $\lceil ? \rceil = \{(\bot, \bot)\}$ while $\lfloor ? \rfloor = \mathrm{Term} \times \mathrm{Term}$.

This approach for logical relations works in that the logical relation is well-defined and if both sides of the birelation η are well-defined, then $\lceil \tau \rceil_\eta = \lfloor \tau \rfloor_\eta$. The issue is that this fact does not scale to general case of $c : \kappa$. In particular, it is not the case that for an $\eta = \eta^{\mathsf{op}}$ that $\lceil \lambda\alpha{:}\kappa.\, c \rceil_\eta = \lfloor \lambda\alpha{:}x.\, \kappa \rfloor_\eta$. The issue is that these must be equal as functions on pairs of relations (R_1, R_2) and not merely on a single relation R. If $R_1 = R_2$ they are equal but this information is not sufficient to get the case for $\forall$ to go through in the fundamental theorem. This failure will be instructive for the coming issue: $\forall$ is the only case where the higher-typed constructors factor in to the expression language. This interaction is crucial for Pottier [46] so it cannot be ignored, but it will be at the heart of the failure of the next attempt.

Instead of considering full recursive kinds, we restrict our attention to a particular class of them: the class of guarded recursive kinds. Guarded recursion in general originates with Nakano [40]. The idea is to isolate which recursive definitions are *productive*: productivity is informally the idea that after a finite number of steps something must be produced. The program need not compute to a normal form but it should produce some observable output. The classic example of productivity is a stream: an infinite stream should never evaluate to a normal form but we would like to be able to calculate the nth element of the stream in finite time. Productive programming is an extremely natural paradigm for many programs which are never expected to terminate but still expected to produce work as they go.

The question is how to ensure that a definition is productive. One line of work is to consider a syntactic condition on programs. This is used by some proof assistants [38] and is simple and sound. The idea is to check that no recursive calls occur without being underneath a constructor. In the stream example, for instance, this means that an element of the stream has to be produced before the function is entitled to recurse. The issue is that a syntactic check is too brittle. It does not, for instance, work if a productive definition has been factored into two different parts since one of these parts may not be syntactically unguarded. It is also ill-suited for handling higher-order functions where the productivity may rely on the supplied function being productive.

A more robust solution is to follow the approach of Nakano [40] and equip the type theory with a modality indicating that something is only available to use "at a later point in time". Then, for instance, in a language with streams the constructor for a stream would have the type $\tau \to \blacktriangleright\mathrm{Stream}_\tau \to \mathrm{Stream}_\tau$ indicating that it is sufficient to provide the tail of a list so that it is only available later. In order to make this modality useful, $\blacktriangleright$ comes equipped with a number of operations. Two of the basic ones ensure that $\blacktriangleright$ is an applicative functor:

$$\mathsf{next}- : \tau \to \blacktriangleright\tau \tag{1}$$

$$- \circledast - : \blacktriangleright(\tau_1 \to \tau_2) \to \blacktriangleright\tau_1 \to \blacktriangleright\tau_2 \tag{2}$$

Finally, $\blacktriangleright$ can be used to construct fixed points.

$$\mathsf{fix} : (\blacktriangleright\tau \to \tau) \to \tau \tag{3}$$

All of this is extremely similar to the operations available in $\widehat{\omega}$. The operations 1 and 2 were facts about the functor $\blacktriangleright$ and 3 is Löb induction. Nakano's work on guarded recursion provides a syntactic account of this modality. In the original presentation the operations written above were presented as subtyping relations in the calculus rendering them all silent. In more recent presentations [9, 17, 19, 20] these operations are all made explicit again as they are mediated by nontrivial isomorphisms. For our purposes, we will continue the subtyping presentation for consistency with Pottier [46] which describes the encoding motivating this construction.

Guarded recursive kinds are slightly different than guarded recursion in general because the term language contains divergent and nonproductive terms. The guarantees provided by guardedness apply only at the level of constructors where it is the case that all constructors are productive (have a weak head normal form). In this sense, the language of guarded recursive kinds can be seen as a particular simply-typed calculus with guarded recursion and a distinguished base type T. The term-formers of this type are the type-formers of our expression language: $- \to -$ and $\forall\alpha{:}\kappa.\,-$. A further distinction between this calculus and other type theories is that our type formers are all contractive. This means that they take arguments available only later and produce a result available now. In type theories with a universe, for instance, the operations on the universe are all nonexpansive: they preserve $\blacktriangleright$s by taking arguments available later and producing a result later (the inverse of the $- \circledast -$ operation would do this for instance). This extra strength provided by contractiveness is exactly what allows for the construction of divergent terms in our language. This would present a soundness issue for a type theory but from our perspective it is a natural convenience of a programming language unconcerned with serving as a logic.

To make this discussion precise, the (selected) rules of our language are the following.

$$\boxed{\Omega \vdash \kappa : \mathsf{kind}}$$

$$\frac{j \in \Omega}{\Omega \vdash j : \mathsf{kind}} \qquad \frac{\Omega \vdash \kappa : \mathsf{kind} \qquad \Omega \vdash \kappa' : \mathsf{kind}}{\Omega \vdash \kappa \to \kappa' : \mathsf{kind}} \qquad \frac{\Omega \vdash \kappa : \mathsf{kind}}{\Omega \vdash \blacktriangleright\kappa : \mathsf{kind}}$$

$$\frac{}{\Omega \vdash \mathsf{T} : \mathsf{kind}} \qquad \frac{\Omega, j : \mathsf{kind} \vdash \kappa : \mathsf{kind} \qquad j.\,\kappa\ \mathsf{guarded}}{\Omega \vdash \mu j.\,\kappa : \mathsf{kind}}$$

$$\boxed{j.\,\kappa\ \mathsf{guarded}}$$

$$\frac{j \text{ not free in } \kappa}{j.\,\kappa\ \mathsf{guarded}} \qquad \frac{j.\,\kappa_1\ \mathsf{guarded} \qquad j.\,\kappa_2\ \mathsf{guarded}}{j.\,\kappa_1 \to \kappa_2\ \mathsf{guarded}} \qquad \frac{}{j.\,\blacktriangleright\kappa\ \mathsf{guarded}}$$

$$\frac{j.\,\kappa\ \mathsf{guarded}}{j.\,(\mu j'.\,\kappa)\ \mathsf{guarded}}$$

$$\boxed{\kappa \leq \kappa'}$$

$$\frac{\kappa_1' \leq \kappa_1 \qquad \kappa_2 \leq \kappa_2'}{\kappa_1 \to \kappa_2 \leq \kappa_1' \to \kappa_2'} \qquad \frac{\kappa \leq \kappa'}{\blacktriangleright\kappa \leq \blacktriangleright\kappa'} \qquad \frac{}{\kappa \leq \blacktriangleright\kappa} \qquad \frac{}{\blacktriangleright(\kappa \to \kappa') \leq \blacktriangleright\kappa \to \blacktriangleright\kappa'}$$

$$\frac{}{\blacktriangleright\kappa \to \blacktriangleright\kappa' \leq \blacktriangleright(\kappa \to \kappa')}$$

$$\boxed{\Delta \vdash c : \kappa \qquad \Delta; \Gamma \vdash e : \tau}$$

$$\frac{\Delta \vdash \tau_1 : \blacktriangleright\mathsf{T} \qquad \Delta \vdash \tau_2 : \blacktriangleright\mathsf{T}}{\Delta \vdash \tau_1 \to \tau_2 : \mathsf{T}} \qquad \frac{\Delta, \alpha : \kappa \vdash \tau : \blacktriangleright\mathsf{T}}{\Delta \vdash \forall\alpha{:}\kappa.\,\tau : \mathsf{T}}$$

$$\frac{\Delta \vdash \tau : \blacktriangleright^n\mathsf{T} \qquad \Delta; \Gamma, x : \tau \vdash e : \tau'}{\Delta; \Gamma \vdash \lambda x{:}\tau.\,e : \tau \to \tau'} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha : \kappa.\tau \qquad \Delta \vdash c : \blacktriangleright^n\kappa}{\Delta; \Gamma \vdash e[c] : [c/\alpha]\tau}$$

Turning now to the question of how to construct a logical relation for this language there are two questions to be answered:

1. We want to interpret kinds as objects of a category. Which category should be used?

2. Pairs of related terms should then be points of this object. What does it mean to be a point of $[\![\mathsf{T}]\!]$.

This two-layer approach is typical of a logical relation for languages with a rich kind structure. We need to interpret kinds as some sort of semantic object so that $[\![\mathsf{T}]\!]$ is the collection of semantic types needed for the logical relation. Traditionally the literature around logical relations for higher-kinded languages has been concerned with notions of parametricity at higher kind [8, 26, 33, 48, 55], which extends the object-relational interpretation of Reynolds [47] to include natural notions of candidates at higher kind. These candidates include more complex conditions that also necessitate the use of semantic tools to handle but it is not the goal of our work. In particular, parametricity arguments will only be interesting for type variables of kind T. At all other kinds the candidates are not relations between semantic types but rather just semantic types. For instance the interpretation of a constructor function of kind $\mathsf{T} \to \mathsf{T}$ is a map between relations, not a relation between maps of semantic types.

This is an undesirable simplification for a general purpose logical relation but the only motivation for constructing this logical relation is to study the semantics of guarded recursive kinds and their impact on the definition. The technical complications introduced by studying reflexive graphs would be wasted when we plan to move to a setting with only T and Kripke worlds. It is not clear what it would mean to consider a model where Kripke worlds were subject to a relational interpretation as well. Such work is latent in work like Dreyer et al. [25] where the Kripke world is more relational to allow for relations between heaps that are not merely point-wise.

Now the answer to these questions is given by the earlier remark that our language is the guarded simply-typed lambda calculus with guarded recursive types and a strange base type. Models of this language are readily available in $\widehat{\omega}$ [15, 17] and we make use of these.

Specifically, we interpret kinds as follows as maps a functor from environments to presheaves. These environments are mixed-variance. They send a kind variable to two distinct presheaves, one used covariantly and one contravariantly. Let us signify these environments by θ (to distinguish them from the environment used by the interpretation of constructors) and write θ^{op} for the operation exchanging all the components of θ. Finally, by convention let $\theta(\alpha)$ project out the presheaf being used covariantly.

$$\begin{aligned}
[\![j]\!]_\theta &= \theta(j) \\
[\![\blacktriangleright\kappa]\!]_\theta &= \blacktriangleright[\![\kappa]\!]_\theta \\
[\![\kappa_1 \to \kappa_2]\!]_\theta &= [\![\kappa_2]\!]_\theta^{[\![\kappa_1]\!]_{\theta^{\mathsf{op}}}} \\
[\![\mu j.\, \kappa]\!]_\theta &= \mathsf{fix}((P_1, P_2) \mapsto [\![\kappa]\!]_{\theta[j \mapsto (P_1, P_2)]}) \\
[\![\mathsf{T}]\!]_\theta &= \mathbb{T}
\end{aligned}$$

It is a straightforward proof to show that this defines a functor. The only exceptional fact is that it is necessary to show that taking a fixed point of some arguments of a mixed-variance functor results in a functor in the remaining arguments. This is proven in Section 7 of Birkedal et al. [17] for instance.

The interesting part of this definition is the interpretation of T, $[\![\mathbb{T}]\!]$. The other clauses are simply induced by the logical meaning of the connectives. Interpreting $\to$ as the exponential, for instance, induces a canonical interpretation of $\lambda\alpha{:}\kappa.\, c$ and $c_1\ c_2$.

The space of semantic types is where we make crucial use of syntactic minimal invariance. First we need to define a class of relations appropriate for semantic types.

Definition 6.1. *A uniform relation is a subset $R \subseteq \mathrm{Term}_{\tau_1} \times \mathrm{Term}_{\tau_2}$ for some τ_1 and τ_2 so that the following hold.*

- *R is closed on both sides under contextual equivalence.*
- *R is admissible and in particular contains limits of all π-chains.*
- *If $(e_1, e_2) \in R$ then for all n, $(\pi^n_{\tau_1}\ e_1, \pi^n_{\tau_2}\ e_2) \in R$.*

The set of uniform relations is denoted URel.

Uniform relations are a natural setting for our logical relation since the assumptions imposed on them are quite natural for anything to be called a semantic type. Items 1 and 3 require that we respect observation in and item 2 is a technically necessary and quite reasonable assumption. The definition of projection in this language is identical to what it is was for the System F-like language considered previously with the slight proviso that instead of $\pi^{n+1}_\alpha = \lambda x{:}\alpha.\, x$ we must generalize it to $\pi^{n+1}_{K[\alpha]} = \lambda x{:}\alpha.\, x$ so that any *neutral type* is impossible to truncate. This definition is forced; there are no other reasonable moves to make at higher kind since without knowing anything about α no information can be deduced about $K[\alpha]$

The presheaf structure on uniform relations is an instance of a more general and well-known embedding of complete 1-bounded ultrametric spaces into $\widehat{\omega}$ described for instance in Birkedal et al. [17]:

$$\mathbb{T}(n) = \mathrm{URel}/{\equiv_n}$$

The equivalence relation at n is defined to equate uniform relations which agree on n truncations. Formally, this is expressed as follows.

$$\begin{aligned} R \equiv_n S = {} & \\ & (\forall (e_1, e_2) \in R.\ (\pi^i_{\tau_1}\ e_1, \pi^i_{\tau_2}\ e_2) \in S) \wedge \\ & (\forall (e_1, e_2) \in S.\ (\pi^i_{\tau_1}\ e_1, \pi^i_{\tau_2}\ e_2) \in R) \end{aligned}$$

Restriction maps simply send equivalence classes of uniform relations to even coarser equivalence classes of uniform relations.

This definition implies that a global point, $h : 1 \to \mathbb{T}$, precisely in correspondence to a particular uniform relation. It picks out a family of equivalence classes so that each is increasingly fine $h(0)(\star) \supseteq h(1)(\star) \supseteq \ldots \supseteq h(n)(\star)$. Syntactic minimal invariance and closure of uniform relation under π-chains tells us that if two uniform relations are equated by $\equiv_n$ for all n then they are equal on the nose. This means that there can be at most one uniform relation in all the equivalence classes selected by h and h is uniquely induced by the uniform relation. This result is in fact a corollary of the fact that the aforementioned embedding preserves finite limits.

Now the logical relation (the constructor interpretation) is really a standard interpretation of the simply-typed lambda calculus with a base type of relations. It is defined as follows.

$$\begin{aligned}
[\![\alpha]\!]_\eta(n) &= \eta(\alpha)(n) \\
[\![\lambda\alpha{:}\kappa.\, c]\!]_\eta(n) &= (\star, S) \mapsto [\![c]\!]_{\eta[\alpha\mapsto S]}(n) \\
[\![c_1\ c_2]\!]_\eta(n) &= [\![c_1]\!]_\eta(n)(\star)([\![c_2]\!]_\eta(n)) \\
[\![\mathsf{out}\, c]\!]_\eta &= \mathsf{out}_n([\![c]\!]_\eta(n)) \\
[\![\mathsf{in}\, c]\!]_\eta &= \mathsf{in}_n([\![c]\!]_\eta(n)) \\
[\![\tau_1 \to \tau_2]\!]_\eta(n) &= \\
&\{(e_1, e_2) \mid e_1 \Downarrow \iff e_2 \Downarrow \wedge \\
&\qquad \forall (a_1, a_2) \in [\![\tau_1]\!]_\eta(n).\ (e_1\ a_1, e_2\ a_2) \in [\![\tau_2]\!]_\eta\} \\
[\![\forall\alpha{:}\kappa.\, \tau]\!]_\eta &= \\
&\{(e_1, e_2) \mid e_1 \Downarrow \iff e_2 \Downarrow \wedge \\
&\qquad \forall c_1, c_2 : \kappa,\ S \in [\![\kappa]\!].\ (e_1[c_1], e_2[c_2]) \in [\![\tau]\!]_{\eta[\alpha\mapsto S]}\}
\end{aligned}$$

In this definition we are using out and in as the isomorphisms mediating a recursive type and its unfolding. There is a technical issue with this presentation: it properly should be done on *derivations* of terms and a coherence theorem must be proven. This is because while it is the case that if $\kappa_1 \leq \kappa_2$ then there is a canonical $i : [\![\kappa_1]\!] \rightarrowtail [\![\kappa_2]\!]$, it is not the case that i is an identity. This means that when the subtyping rule is applied in a derivation the resulting semantic object must be adjusted explicitly. This issue is obliviated when working in ultrametric spaces because the subtyping coercions are simply identities there. This technical detail is not relevant to the issue with this approach and thus we have suppressed it.

The extra applications of n and $\star$ are results of the fact that the interpretation is a natural transformation from $[\![\Delta]\!]$, here written η, to $[\![\kappa]\!]$. This is defined by an indexed family of functions which is what has been done here. Additionally, since constructor functions are exponentials in the category at stage n they are natural transformations $\mathsf{y}(n) \times [\![\kappa_1]\!]_\eta \to [\![\kappa_2]\!]_\eta$ so we must apply them to an element of $\mathsf{y}(n)(n) = \{\star\}$ in addition to the actual argument.

The issue arises from the fact that this must be made well-typed and natural. These two requirements are inexorably tied together by the interpretation of λ which is only a valid natural transformation if $[\![c]\!]$ is natural in η. It is unfortunately the case that these properties fail to hold for the definition of $[\![\forall\alpha{:}\kappa.\, \tau]\!]$. In this case, being well-typed implies that for at n, for every pair η,

η' which are equal at stage n it must be that $[\![\forall\alpha{:}\kappa.\,\tau]\!]_\eta \equiv_n [\![\forall\alpha{:}\kappa.\,\tau]\!]_{\eta'}$. Importantly, equality at stage n is *not* necessarily just pointwise equality because two semantic types are equal at stage n if they are related by $\equiv_n$. This essentially is a statement that $[\![\forall\alpha{:}\kappa.\,\tau]\!]$ is functional with respect to $\equiv_n$ and this is simply false.

As a counterexample, consider the following instance of τ.

$$\tau = \alpha\ \beta \to \alpha\ \gamma$$

For convenience, let us denote $\forall\alpha{:}\mathsf{T} \to \mathsf{T}.\,\alpha\ \beta \to \alpha\ \gamma$ as τ'. In this case we must show that show that $[\![\tau']\!]_\eta \equiv_n [\![\tau']\!]_{\eta'}$ provided that $\eta(\beta) \equiv_n \eta'(\beta)$ and $\eta(\gamma) \equiv_n \eta'(\gamma)$. Let us suppose that $\eta(\beta)$ is a uniform relation $R \subseteq \tau_a \times \tau_a$ and $\eta'(\beta)$ is $S \subseteq \tau_a \times \tau_a$ and both $\eta(\gamma)$ and $\eta'(\gamma)$ are R. Assume that $R \neq S$. If such R and S cannot be constructed then URel is trivial and our logical relation is degenerate.

We are required to show that π^n transports from $[\![\tau']\!]_\eta(n)$ to $[\![\tau']\!]_{\eta'}(n)$ and back again. Proving the first part of this means showing that if $(e_1, e_2) \in [\![\tau']\!]_\eta(n)$ then the following holds.

$$(\pi_{[\tau_a/\beta]\tau'}\ e_1, \pi_{[\tau_a/\beta]\tau'}\ e_1) \in [\![\tau']\!]_{\eta'}(n)$$

However, if $n > 2$ and e_1 and e_2 do not diverge, then this is equivalent to showing the following.

$$(\Lambda\alpha{:}\mathsf{T}.\,e_1[\alpha], \Lambda\alpha{:}\mathsf{T}.\,e_1[\alpha]) \in [\![\tau']\!]_{\eta'}(n)$$

This can be simplified further since $[\![\tau']\!]_{\eta'}(n)$ is a uniform relation to just the following.

$$(e_1, e_2) \in [\![\tau']\!]_{\eta'}(n)$$

Which is to say, we are required to show that if η and η' are only related by $\equiv_n$ then we must show that $[\![\tau']\!]_\eta$ and $[\![\tau']\!]_{\eta'}$. However, it is not hard to show that the following two facts are true.

$$(\Lambda\alpha{:}\mathsf{T} \to \mathsf{T}.\,\lambda x{:}\alpha\ \tau_a.\,x, \Lambda\alpha{:}\mathsf{T} \to \mathsf{T}.\,\lambda x{:}\alpha\ \tau_a.\,x) \in [\![\tau']\!]_\eta$$

$$(\Lambda\alpha{:}\mathsf{T} \to \mathsf{T}.\,\lambda x{:}\alpha\ \tau_a.\,x, \Lambda\alpha{:}\mathsf{T} \to \mathsf{T}.\,\lambda x{:}\alpha\ \tau_a.\,x) \notin [\![\tau']\!]_{\eta'}$$

This means that $[\![-]\!]$ is not type-correct and not a well-formed definition. At the root of the issue is the fact that syntactic minimal invariance does not seem to scale to higher kinds without some fundamentally new idea. Without something akin to a type-casing operation it does not seem possible to define a correct version of π at neutral types. It really seems that it must proceed by induction on the variable the neutral constructor is stuck on. This issue does not arise in domain theory because in this setting projections are untyped and case on the structure of the element of the domain not the "type" that this element may later be shown to inhabit.

Any cure for this issue seems worse than the illness. Adding type-case for instance allows us to define a correct projection for instance but it destroys the parametricity of the system. One could consider a system like LX [23] and define projection with respect to the reification of a type. This solves the issue for all the types that we have reifications of, but there is a diagonalization issue here: codes are needed for every single constructor but the codes themselves are constructors.

Finally, as mentioned previously URel is inadequate for representation independence results. Since we have required that uniform relations are closed under truncations any uniform relation between different types will relate divergent terms on one side to nondivergent terms on the other. This puts a complicates using technique to construct a logical relation for state. Essentially, we have removed the explicitly need for trunc_i but all proofs must act as though trunc_i is still in the language. All together, this means that even fixing this particular logical relation is likely inadequate for our purposes.

7 Conclusions

This thesis has illustrated the difficulty of constructing a logical relation for a language with higher-order state. The current set of techniques which work are complex and despite this they only provide sound but incomplete reasoning principles. Even with this logical relations are among the best known techniques for handling effects. They strike a combination between using simpler mathematics and providing useful reasoning principles.

The difficulty and complexity of the attempts to remove step-indexing from the logical relation raises a question: is it worth it? The motivations for step-index-free logical relations were basically the following:

- Logical relations for step-indexing were complicated and painful to use and construct correctly.
- Step-indexing is ad-hoc and difficult to justify.
- Step-indexing is ill-suited for capturing properties beyond safety.

The first two issues are addressed in the same way: a generalization in the way we view logical relations. If logical relations are just structure-preserving relations, there's no reason to limit ourselves to traditional set-theoretic relations. Already with Kripke logical relations there is the same indexing structure. Rather than viewing relations as holding or not holding it is quite natural to replace them with a more internsional, local notion of truth as is done in both step-indexing and Kripke logical relations. Mathematically, this is just a switch from relations in sets to relations in an appropriate presheaf category. If we make use of more sophisticated tools for working with these relations, either by working internally to a type theory, a higher-order logic, or a topos much of the day-to-day issues of step-indexing can be resolved.

The last issue is a real problem. The nature of step-indexing is inherently limiting because it is only is suited for properties that are sound with respect to finite prefixes. Consider for instance, the set of programs producing a stream of natural numbers. The program outputting 1, 2, 3, etc. At any finite step this program produces a maximal number but it is not the case that it ever actually outputs a maximum globally. This is a classic issue in mathematics: global truth is not truth in every localization and step-indexing only allows us to discuss local truth.

In order to fix this, one can either remove the indexing structure and never work with local truth or add enough locales so that local truth is strong enough. For instance, if we work with ω^2 in the example above the problematic program is ruled out: it does not satisfy the program at $(\omega, 0)$. A fuller theory of logical relation in this setting is given by Svendsen et al. [52].

Further investigation is needed in both directions. There are many unstudied generalizations of relations beyond merely a set-theoretic subset of a product. On the other hand, there is an undeniable appeal to just being able to work with sets and to find some new clever technique to avoid the indexing all together. At the moment though, there seems to be far more unexplored and accessible space in the first direction than the second; constructing a step-index-free logical relation appears to require some new approach beyond applications of classic tools as was done in this thesis.

Acknowledgments

I would like to thank the huge number of people who have been instrumental in writing this thesis and the research of the last four years that went into it. I am greatly indebted to the POP group at CMU for a unique environment to study. A special thanks to Carlo Angiuli, Evan Cavallo, Adrien Guatto, Anders Mörtberg, Jonathan Sterling, and Joseph Tassarotti for countless hours of discussions on programming languages. In particular, I must thank Jonathan Sterling and Carlo Angiuli for valuable feedback on a draft of this thesis. Thanks to Lars Birkedal and Robert Harper for feedback and advice on this work. Finally, I must thank Karl Crary for advice, encouragement, and much wisdom into how to do research in type theory.

References

[1] Martín Abadi and Gordon D. Plotkin. A per model of polymorphism and recursive types. In *Logic in Computer Science*, 1990.

[2] Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*. 1994.

[3] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*, 2006.

[4] Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Department of Computer Science, Princeton University, 2004.

[5] Pierre America and Jan J. M. M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. In *Workshop on Mathematical Foundations of Programming Language Semantics*, 1988.

[6] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions Programming Language Systems*, 2001.

[7] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *Principles of Programming Languages*, 2007.

[8] Robert Atkey. Relational parametricity for higher kinds. In *Computer Science Logic*, 2012.

[9] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The Clocks Are Ticking: No More Delays! In *Logic in Computer Science*, 2017.

[10] E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 1990.

[11] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2013.

[12] Lars Birkedal and Robert Harper. Constructing interpretations of recursives types in an operational setting. *Information and Computation*, 1999.

[13] Lars Birkedal and Rasmus Møgelberg. Categorical models for Abadi and Plotkins logic for parametricity. In *Mathematical Structures in Computer Science*, 2005.

[14] Lars Birkedal and Rasmus Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *Logic in Computer Science*, 2013.

[15] Lars Birkedal, Jan Schwinghammer, and Kristian Støvring. A metric model of lambda calculus with guarded recursion. In *Fixed Points in Computer Science*, 2010.

[16] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theoretical Computer Science*, 2010.

[17] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *Logic in Computer Science*, 2011.

[18] Lars Birkedal, Kristen Støvring, and Jacob Thamsborg. A relational realizability model for higher-order stateful ADTs. *Logic and Algebraic Programming*, 2012.

[19] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Gratwohl, Bas Spitters, and Andrea Vezzosi. Guarded cubical type theory. In *TYPES*, 2016.

[20] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In *Foundations of Software Science and Computation Structures*, 2016.

[21] Karl Crary. Modules, abstraction, and parametric polymorphism. In *Principles of Programming Languages*, 2017.

[22] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electronic Notes Theoretical Computer Science*, 2007.

[23] Karl Crary and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, 1999.

[24] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *Logic In Computer Science*, 2009.

[25] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *International Conference on Functional Programming*, 2010.

[26] Brian Dunphy and Uday Reddy. Parametric limits. In *Logic in Computer Science*, 2004.

[27] Martín Escardó. A metric model of pcf. In *Workshop on Realizability Semantics and Applications*, 1998.

[28] Peter Freyd. Algebraically complete categories. In *Category Theory*, 1970.

[29] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, D. S. Scott, and G.-C. Rota. *Continuous Lattices and Domains*. 2003.

[30] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, 1971.

[31] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[32] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[33] Ryu Hasegawa. Categorical data types in parametric polymorphism. 1994.

[34] Claudio Hermida, Uday Reddy, and Edmund Robinson. Logical relations and parametricity a reynolds programme for category theory and programming languages. 2014.

[35] Martin Hofmann and Thomas Steicher. Lifting grothendieck universes, 199? Unpublished note, available online at `http://www.mathematik.tudarmstadt.de/~streicher/NOTES/lift.dvi.gz`.

[36] M. Krogh-Jespersen, K. Svendsen, and L. Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *Principles of Programming Languages*, 2017.

[37] QingMing Ma and John C. Reynolds. Types, abstractions, and parametric polymorphism, part 2. In *Mathematical Foundations of Programming Semantics*, 1991.

[38] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`.

[39] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 1991.

[40] Hiroshi Nakano. A modality for recursion. In *Logic in Computer Science*, 2000.

[41] M. Paviotti, R.E. Møgelberg, and L. Birkedal. A model of PCF in guarded type theory. In *Mathematical Foundations of Programming Semantics*, 2015.

[42] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 1996.

[43] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

[44] Andrew M. Pitts and Ian D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. 1998.

[45] Gordon Plotkin. Lambda-definability in the full type hierarchy. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980.

[46] François Pottier. A typed store-passing translation for general references. In *Principles of Programming Languages*, 2011.

[47] John C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing*, 1983.

[48] Edmund Robinson and Giuseppe Rosolini. Reflexive graphs and parametric polymorphism. In *Logic in Computer Science*, 1994.

[49] Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 1976.

[50] Michael Smyth and Gordon Plotkin. The category-theoretic solution of recursive domain equations. In *SIAM Journal on Computing*, 1977.

[51] Thomas Streicher. Universes in toposes, 2004. Unpublished note, available online at `http://www.mathematik.tu-darmstadt.de/~streicher/NOTES/UniTop.pdf`.

[52] Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. Transfinite step-indexing: Decoupling concrete and logical steps. In *European Symposium on Programming*, 2016.

[53] William W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 1967.

[54] Aaron J. Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *Principles of Programming Languages*, 2013.

[55] Dimitrios Vytiniotis and Stephanie Weirich. Parametricity, type equality, and higher-order polymorphism. In *Journal of Functional Programming*, 2010.