Denotational Semantics for Gradual Typing in Synthetic Guarded Domain Theory

ERIC GIOVANNINI and MAX S. NEW

We develop a denotational semantics for a gradually typed language with effects that is adequate and proves the graduality theorem. The denotational semantics is constructed using *synthetic guarded domain theory* working in a type theory with a later modality and clock quantification. This provides a remarkably simple presentation of the semantics, where gradual types are interpreted as ordinary types in our ambient type theory equipped with an ordinary preorder structure to model the error ordering. This avoids the complexities of classical domain-theoretic models (New and Licata) or logical relations models using explicit step-indexing (New and Ahmed). In particular, we avoid a major technical complexity of New and Ahmed that requires two logical relations to prove the graduality theorem.

By working synthetically we can treat the domains in which gradual types are interpreted as if they were ordinary sets. This allows us to give a "naïve" presentation of gradual typing where each gradual type is modeled as a well-behaved subset of the universal domain used to model the dynamic type, and type precision is modeled as simply a subset relation.

ACM Reference Format:
Eric Giovannini and Max S. New. 2018. Denotational Semantics for Gradual Typing in Synthetic Guarded Domain Theory. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

1.1 Gradual Typing and Graduality

Gradual typing allows a language to have both statically-typed and dynamically-typed terms; the statically-typed terms are type checked at compile time, while type checking for the dynamically-typed terms is deferred to runtime.

Gradually-typed languages should satisfy two intuitive properties. First, the interaction between the static and dynamic components of the codebase should be safe – i.e., should preserve the guarantees made by the static types. In other words, in the static portions of the codebase, type soundness must be preserved. Second, gradual langugaes should support the smooth migration from dynamic typing to static typing, in that the programmer can initially leave off the typing annotations and provide them later without altering the meaning of the program.

Formally speaking, gradually typed languages should satisfy the *dynamic gradual guarantee*, originally defined by Siek, Vitousek, Cimini, and Boyland [3]. This property is also referred to as *graduality*, by analogy with parametricity. Intuitively, graduality says that in going from a dynamic to static style should not introduce changes in the meaning of the program. More specifically, making the types more precise by adding annotations will either result in the same behavior as the less precise program, or result in a type error.

Woodstock '18, June 03–05, 2018, Woodstock, NY

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

1.2 Current Approaches

Current approaches to proving graduality include the methods of Abstracting Gradual Typing [2] and the formal tools of the Gradualier [1]. These allow the language developer to start with a statically typed langauge and derive a gradually typed language that satisfies the gradual guarantee. The downside is that not all gradually typed languages can be derived from these frameworks, and moreover, in both approaches the semantics is derived from the static type system as opposed to the alternative in which the semantics determines the type checking. Without a clear semantic interpretation of type dynamism, it becomes difficult to extend these techniques to new language features such as polymorphism.

New and Ahmed have developed a semantic approach to specifying type dynamism in term of embedding-projection pairs, which allows for a particularly elegant formulation of the gradual guarantee. Moreover, this approach allows for type-based reasoning using η-equivalences

The downside of the above approach is that each new language requires a different logical relation to prove graduality. As a result, many developments using this approach require vast effort, with many such papers having 50+ pages of proofs. Our aim here is that by mechanizing a graduality proof in a reusable way, we will provide a framework for other language designers to use to ensure that their languages satsify graduality.

One approach currently used to prove graduality uses the technique of *logical relations*. Specifially, a logical relation is constructed and shown to be sound with respect to observational approximation. Because the dynamic type is modeled as a non-well-founded recursive type, the logical relation needs to be paramterized by natural numbers to restore well-foundedness. This technique is known as a *step-indexed logical relation* [**?**]. Reasoning about step-indexed logical relations can be tedious and error-prone, and there are some very subtle aspects that must be taken into account in the proofs. Figure **??** shows an example of a step-indexed logical relation for the gradually-typed lambda calculus.

In particular, the prior approach of New and Ahmed requires two separate logical relations for terms, one in which the steps of the left-hand term are counted, and another in which the steps of the right-hand term are counted [**?**]. Then two terms M and N are related in the "combined" logical relation if they are related in both of the one-sided logical relations. Having two separate logical relations complicates the statement of the lemmas used to prove graduality, becasue any statement that involves a term stepping needs to take into account whether we are counting steps on the left or the right. Some of the differences can be abstracted over, but difficulties arise for properties as fundamental and seemingly straightforward as transitivty.

Specifically, for transitivity, we would like to say that if M is related to N at index i and N is related to P at index i, then M is related to P at i. But this does not actually hold: we requrie that one of the two pairs of terms be related "at infinity", i.e., that they are related at i for all $i \in \mathbb{N}$. Which pair is required to satisfy this depends on which logical relation we are considering, (i.e., is it counting steps on the left or on the right), and so any argument that uses transitivity needs to consider two cases, one where M and N must be shown to be related for all i, and another where N and P must be related for all i. This may not even be possible to show in some scenarios!

These complications introduced by step-indexing lead one to wonder whether there is a way of proving graduality without relying on tedious arguments involving natural numbers. An alternative approach, which we investigate in this paper, is provided by *synthetic guarded domain theory*, as discussed below. Synthetic guarded domain theory allows the resulting logical relation to look almost identical to a typical, non-step-indexed logical relation.

1.3 Contributions

Our main contribution is a framework in Guarded Cubical Agda for proving graduality of a cast calculus. To demonstrate the feasability and utility of our approach, we have used the framework to prove graduality for the simply-typed gradual lambda calculus. Along the way, we have developed an intensional theory of graduality that is of independent interest.

1.4 Proving Graduality in SGDT

In a gradually-typed language, the mixing of static and dynamic code is seamless, in that the dynamically typed parts are checked at runtime. This type checking occurs at the elimination forms of the language (e.g., pattern matching, field reference, etc.). Gradual languages are generally elaborated to a *cast calculus*, in which the dynamic type checking is made explicit through the insertion of *type casts*.

In a cast calculus, there is a relation $\sqsubseteq$ on types such that $A \sqsubseteq B$ means that A is a *more precise* type than B. There a dynamic type ? with the property that $A \sqsubseteq ?$ for all A. If $A \sqsubseteq B$, a term M of type A may be *up*casted to B, written $\langle B \nwarrow A \rangle M$, and a term N of type B may be *down*casted to A, written $\langle A \swarrow B \rangle N$. Upcasts always succeed, while downcasts may fail at runtime. We also have a notion of *syntatcic term precision*. If $A \sqsubseteq B$, and M and N are terms of type A and B respectively, we write $M \sqsubseteq N : A \sqsubseteq B$ to mean that M is more precise than N, i.e., M and N behave the same except that M may error more.

In this paper, we will be using SGDT techniques to prove graduality for a particularly simple gradually-typed cast calculus, the gradually-typed lambda calculus. This is just the usual simply-typed lambda calculus with a dynamic type ? such that $A \sqsubseteq ?$ for all types A, as well as upcasts and downcasts between any types A and B such that $A \sqsubseteq B$. The complete definition will be provided in Section 3.

Our main theorem is the following:

Theorem 1.1 (Graduality). *If* $\cdot \vdash M \sqsubseteq N : \mathit{Nat}$, *then*

(1) If $N = \mathfrak{U}$, *then* $M = \mathfrak{U}$
(2) If $N = \text{`}n$, *then* $M = \mathfrak{U}$ *or* $M = \text{`}n$
(3) If $M = V$, *then* $N = V$

We also should be able to show that $\mathfrak{U}$, zro, and suc N are not equal.

Our first step toward proving graduality is to formulate an *intensional* gradual lambda calculus, which we call Int-λC, in which the computation steps taken by a term are made explicit. The "normal" graudal lambda calculus for which we want to prove graduality will be called the *extensional* gradual lambda calculus, denoted Ext-λC. We will define an erasure function $\lfloor \cdot \rfloor$: Int-$\lambda C \to$ Ext-λC which takes a program in the intensional lambda calculus and "forgets" the syntactic information about the steps to produce a term in the extensional calculus.

Every term M_e in Ext-λC will have a corresponding program M_i in Int-λC such that $\lfloor M_i \rfloor = M_e$. Moreover, we will show that if $M_e \sqsubseteq_e M'_e$ in the extensional theory, then there exists terms M_i and M'_i such that $\lfloor M_i \rfloor = M_e$, $\lfloor M'_i \rfloor = M'_e$ and $M_i \sqsubseteq_i M'_i$ in the intensional theory.

We formulate and prove an analogous graduality theorem for the intensional lambda calculus. We define an interpretation of the intensional lambda calculus into a model in which we prove various results. Using the observation above, given $M_e \sqsubseteq M'_e$: Nat, we can find intensional programs M_i and M'_i that erase to them and are such that $M_i \sqsubseteq M'_i$. We will then apply the intensional graduality theorem to M_i and M'_i, and translate the result back to M_e and M'_e.

1.5 Overview of Remainder of Paper

In Section 2, we provide technical background on gradually typed languages and on synthetic guarded domain theory. In Section 3, we introduce the gradually-typed cast calculus for which we will prove graduality. Important here are the notions of syntactic type precision and term precision. We introduce both the extensional gradual lambda calculus (Ext-λC) and the intensional gradual lambda calculus (Int-λC). In Section 4, we define several fundamental constructions internal to SGDT that will be needed when we give a denotational semantics to our intensional lambda calculus. This includes the notion of Predomains as well as the concept of EP-Pairs. In Section 5, we define the denotational semantics for the intensional gradually-typed lambda calculus using the domain theoretic constructions in the previous section. In Section 6, we outline in more detail the proof of graduality for the extensional gradual lambda calculus. In Section 7, we discuss the benefits and drawbacks to our approach in comparison to the traditional step-indexing approach, as well as possibilities for future work.

2 TECHNICAL BACKGROUND

2.1 Synthetic Guarded Domain Theory

One way to avoid the tedious reasoning associated with step-indexing is to work axiomatically inside of a logical system that can reason about non-well-founded recursive constructions while abstracting away the specific details of step-indexing required if we were working analytically. The system that proves useful for this purpose is called *synthetic guarded domain theory*, or SGDT for short. We provide a brief overview here, but more details can be found in [**?**].

SGDT offers a synthetic approach to domain theory that allows for guarded recursion to be expressed syntactically via a type constructor $\triangleright : \mathtt{Type} \to \mathtt{Type}$ (pronounced "later"). The use of a modality to express guarded recursion was introduced by Nakano [**?**].

Given a type A, the type $\triangleright\, A$ represents an element of type A that is available one time step later. There is an operator $\mathsf{next} : A \to \triangleright\, A$ that "delays" an element available now to make it available later. We will use a tilde to denote a term of type $\triangleright\, A$, e.g., $\tilde{M}$.

There is a fixpoint operator

$$\mathsf{fix} : \forall T, (\triangleright\, T \to T) \to T.$$

That is, to construct a term of type T, it suffices to assume that we have access to such a term "later" and use that to help us build a term "now". This operator satisfies the axiom that $\mathsf{fix} f = f(\mathsf{next}(\mathsf{fix} f))$. In particular, this axiom applies to propositions $P : \mathtt{Prop}$; proving a statement in this manner is known as Löb-induction.

In SGDT, there is also a new sort called *clocks*. A clock serves as a reference relative to which the constructions described above are carried out. For instance, given a clock k and type T, the type $\triangleright^k\, T$ represents a value of type T one unit of time in the future according to clock k. If we only ever had one clock, then we would not need to bother defining this notion. However, the notion of *clock quantification* is crucial for encoding coinductive types using guarded recursion, an idea first introduced by Atkey and McBride [**?**].

2.1.1 Ticked Cubical Type Theory. In Ticked Cubical Type Theory [**?**], there is an additional sort called *ticks*. Given a clock k, a tick $t : \mathsf{tick} k$ serves as evidence that one unit of time has passed according to the clock k. The type $\triangleright\, A$ is represented as a function from ticks of a clock k to A. The type A is allowed to depend on t, in which case we write $\triangleright^k_t\, A$ to emphasize the dependence.

The rules for tick abstraction and application are similar to those of dependent Π types. In particular, if we have a term M of type $\triangleright^k\, A$, and we have available in the context a tick $t' : \mathsf{tick} k$,

then we can apply the tick to M to get a term $M[t'] : A[t'/t]$. We will also write tick application as M_t. Conversely, if in a context $\Gamma, t : \mathsf{tick}k$ we have that M has type A, then in context Γ we have $\lambda t.M$ has type $\triangleright A$.

The statements in this paper have been formalized in a variant of Agda called Guarded Cubical Agda [**?**], an implementation of Clocked Cubical Type Theory.

2.2 The Topos of Trees Model

The topos of trees $\mathcal{S}$ is the presheaf category Set^{ω^o}.

We assume a universe $\mathcal{U}$ of types, with encodings for operations such as sum types (written as $\widehat{+}$). There is also an operator $\widehat{\triangleright} : \triangleright \mathcal{U} \to \mathcal{U}$ such that $\mathsf{El}(\widehat{\triangleright}(\mathsf{next}A)) = \triangleright\, \mathsf{El}(A)$, where El is the type corresponding to the code A.

An object X is a family $\{X_i\}$ of sets indexed by natural numbers, along with restriction maps $r_i^X : X_{i+1} \to X_i$.

A morphism from $\{X_i\}$ to $\{Y_i\}$ is a family of functions $f_i : X_i \to Y_i$ that commute with the restriction maps in the obvious way, that is, $f_i \circ r_i^X = r_i^Y \circ f_{i+1}$.

The type operator $\triangleright$ is defined on an object X by $(\triangleright\, X)_0 = 1$ and $(\triangleright\, X)_{i+1} = X_i$. The restriction maps are given by $r_0^{\triangleright} = !$, where $!$ is the unique map into 1, and $r_{i+1}^{\triangleright} = r_i^X$. The morphism $\mathsf{next}^X : X \to \triangleright\, X$ is defined pointwise by $\mathsf{next}_0^X = !$, and $\mathsf{next}_{i+1}^X = r_i^X$.

Given a morphism $f : \triangleright\, X \to X$, we define $\mathsf{fix}f$ pointwise as $\mathsf{fix}_i(f) = f_i \circ \cdots \circ f_0$.

3 GTLC

Here we describe the syntax and typing for the graudally-typed lambda calculus. We also give the rules for syntactic type and term precision.

We start with the extensional lambda calculus Ext-λC, and then describe the additions necessary for the intensional lambda calculus Int-λC.

3.1 Syntax

$$\begin{array}{l}\text{Types } A, B ::= \mathrm{Nat},\ ?,\ (A \Rightarrow B)\\ \text{Terms } M, N ::= \mathfrak{U}_A,\ \mathrm{zro},\ \mathrm{suc}\, M,\ (\lambda x.M),\ (M\, N),\ (\langle B \nwarrow A \rangle M),\ (\langle A \swarrow B \rangle M)\\ \text{Contexts } \Gamma ::= \cdot,\ (\Gamma, x : A)\end{array}$$

The typing rules are as expected, with a cast between A to B allowed only when $A \sqsubseteq B$.

$$\frac{}{\Gamma \vdash \mathfrak{U}_A : A} \qquad \frac{}{\Gamma \vdash \mathrm{zro} : \mathrm{Nat}} \qquad \frac{\Gamma \vdash M : \mathrm{Nat}}{\Gamma \vdash \mathrm{suc}\, M : \mathrm{Nat}} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \Rightarrow B}$$

$$\frac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B} \qquad \frac{A \sqsubseteq B \qquad \Gamma \vdash M : A}{\Gamma \vdash \langle B \nwarrow A \rangle M : B} \qquad \frac{A \sqsubseteq B \qquad \Gamma \vdash M : B}{\Gamma \vdash \langle A \swarrow B \rangle M : A}$$

The equational theory is also as expected, with β and η laws for function type.

3.2 Type Precision

The rules for type precision are as follows:

$$\frac{}{? \sqsubseteq ?}\ ? \qquad \frac{}{\text{Nat} \sqsubseteq \text{Nat}}\ \textsc{Nat} \qquad \frac{}{\text{Nat} \sqsubseteq ?}\ Inj_{\text{Nat}} \qquad \frac{A_i \sqsubseteq B_i \qquad A_o \sqsubseteq B_o}{(A_i \Rightarrow A_o) \sqsubseteq (B_i \Rightarrow B_o)} \Rightarrow$$

$$\frac{(A_i \to A_o) \sqsubseteq (? \Rightarrow ?)}{(A_i \to A_o) \sqsubseteq ?}\ Inj_{\Rightarrow}$$

Note that as a consequence of this presentation of the type precision rules, we have that if $A \sqsubseteq B$, there is a unique precision derivation that witnesses this. As in previous work, we go a step farther and make these derivations first-class objects, known as *type precision derivations*. Specifically, for every $A \sqsubseteq B$, we have a derivation $d : A \sqsubseteq B$ that is constructed using the rules above. For instance, there is a derivation $? : ? \sqsubseteq ?$, and a derivation $\text{Nat} : \text{Nat} \sqsubseteq \text{Nat}$, and if $d_i : A_i \sqsubseteq B_i$ and $d_o : A_o \sqsubseteq B_o$, then there is a derivation $d_i \Rightarrow d_o : (A_i \Rightarrow A_o) \sqsubseteq (B_i \Rightarrow B_o)$. Likewise for the remaining two rules. The benefit to making these derivations explicit in the syntax is that we can perform induction over them. Note also that for any type A, we use A to denote the reflexivity derivation that $A \sqsubseteq A$, i.e., $A : A \sqsubseteq A$. Finally, observe that for type precision derivations $d : A \sqsubseteq B$ and $d' : B \sqsubseteq C$, we can define their composition $d' \circ d : A \sqsubseteq C$. This will be used in the statement of transitivity of the term precision relation.

3.3 Term Precision

We allow for a *heterogeneous* term precision judgment on terms M of type A and N of type B, provided that $A \sqsubseteq B$ holds.

In order to deal with open terms, we will need the notion of a type precision *context*, which we denote $\Gamma^{\sqsubseteq}$. This is similar to a normal context but instead of mapping variables to types, it maps variables x to related types $A \sqsubseteq B$, where x has type A in the left-hand term and B in the right-hand term. We may also write $x : d$ where $d : A \sqsubseteq B$ to indicate this. Another way of thinking of type precision contexts is as a zipped pair of contexts Γ^l, Γ^r with the same domain such that $\Gamma^l(x) \sqsubseteq \Gamma^r(x)$ for each x in the domain.

The rules for term precision come in two forms. We first have the *congruence* rules, one for each term constructor. These assert that the term constructors respect term precision. The congruence rules are as follows:

$$\frac{\Gamma \vdash M : A}{\Gamma^{\sqsubseteq} \vdash \mho_A \sqsubseteq_e M : A}\ \mho \qquad \frac{d : A \sqsubseteq B \qquad \Gamma^{\sqsubseteq}(x) = (A, B)}{\Gamma^{\sqsubseteq} \vdash x \sqsubseteq_e x : d}\ \textsc{Var} \qquad \frac{}{\Gamma^{\sqsubseteq} \vdash \text{zro} \sqsubseteq_e \text{zro} : \text{Nat}}\ \textsc{Zro}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e N : \text{Nat}}{\Gamma^{\sqsubseteq} \vdash \text{suc}\, M \sqsubseteq_e \text{suc}\, N : \text{Nat}}\ \textsc{Suc}$$

$$\frac{d_i : A_i \sqsubseteq B_i \qquad d_o : A_o \sqsubseteq B_o \qquad \Gamma^{\sqsubseteq}, x : d_i \vdash M \sqsubseteq_e N : d_o}{\Gamma^{\sqsubseteq} \vdash \lambda x.M \sqsubseteq_e \lambda x.N : (d_i \Rightarrow d_o)}\ \textsc{Lambda}$$

$$\frac{\begin{array}{c} d_i : A_i \sqsubseteq B_i \qquad d_o : A_o \sqsubseteq B_o \\ \Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e M' : (d_i \Rightarrow d_o) \qquad \Gamma^{\sqsubseteq} \vdash N \sqsubseteq_e N' : d_i \end{array}}{\Gamma^{\sqsubseteq} \vdash M\, N \sqsubseteq_e M'\, N' : d_o}\ \textsc{App}$$

We then have additional equational axioms, including transitivity, β and η laws, and rules characterizing upcasts as least upper bounds, and downcasts as greatest lower bounds.

We write $M \sqsupseteq\sqsubseteq N$ to mean that both $M \sqsubseteq N$ and $N \sqsubseteq M$.

$$\frac{\Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e N : d \qquad \Gamma^{\sqsubseteq} \vdash N \sqsubseteq_e P : d'}{\Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e P : d' \circ d}\ \textsc{Transitivity} \qquad \frac{\Gamma, x : A_i \vdash M : A_o \qquad \Gamma \vdash V : A_i}{\Gamma^{\sqsubseteq} \vdash (\lambda x.M)\, V \sqsupseteq\sqsubseteq_e M[V/x] : A_o}\ \beta$$

$$\frac{\Gamma \vdash V : A_i \Rightarrow A_o}{\Gamma^{\sqsubseteq} \vdash \lambda x.(V\, x) \sqsupseteq\sqsubseteq_e V : A_i \Rightarrow A_o}\ \eta \qquad \frac{d : A \sqsubseteq B \qquad \Gamma \vdash M : A}{\Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e \langle B \nwarrow A \rangle M : d}\ \textsc{UpR}$$

$$\frac{d : A \sqsubseteq B \qquad \Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e N : d}{\Gamma^{\sqsubseteq} \vdash \langle B \nwarrow A \rangle M \sqsubseteq_e N : B}\ \textsc{UpL} \qquad \frac{d : A \sqsubseteq B \qquad \Gamma \vdash M : B}{\Gamma^{\sqsubseteq} \vdash \langle A \swarrow B \rangle M \sqsubseteq_e M : d}\ \textsc{DnL}$$

$$\frac{d : A \sqsubseteq B \qquad \Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e N : d}{\Gamma^{\sqsubseteq} \vdash M \sqsubseteq_e \langle A \swarrow B \rangle N : A}\ \textsc{DnR}$$

The rules UpR, UpL, DnL, and DnR were introduced in [**?**] as a means of cleanly axiomatizing the intended behavior of casts in a way that doesn't depend on the specific constructs of the language. Intuitively, rule UpR says that the upcast of M is an upper bound for M in that M may error more, and UpL says that the upcast is the *least* such upper bound, in that it errors more than any other upper bound for M. Conversely, DnL says that the downcast of M is a lower bound, and DnR says that it is the *greatest* lower bound.

3.4 The Intensional Lambda Calculus

Now that we have described the syntax along with the type and term precision judgments for Ext-λC, we can now do the same for Int-λC. One key difference between the two calculi is that we define Int-λC using the constructs available to us in the language of synthetic guarded domain theory, e.g., we use the $\triangleright$ operator. Whereas when we defined the syntax of the extensional lambda calculus we were working in the category of sets, when we define the syntax of the intensional lambda calculus we will be working in the topos of trees.

More specifically, in Int-λC, we not only have normal terms, but also terms available "later", which we denote by $\tilde{M}$. We have a term constructor θ_s which takes a later-term and turns it into a term avaialble now. The typing and precision rules for θ_s involve the $\triangleright$ operator, as shown below. Observe that θ_s is a syntactic analogue to the θ constructor of the lifting monad that we will define in the section on domain theoretic constructions (Section 4), but it is important to note that $\theta_s(\tilde{M})$ is an actual term in Int-λC, whereas the θ constructor is a purely semantic construction. These will be connected when we discuss the interpretation of Int-λC into the semantic model.

To better understand this situation, note that $\lambda x.(\cdot)$ can be viewed as a function (at the level of the ambient type theory) from terms of type B under context $\Gamma, x : A$ to terms of type $A \Rightarrow B$ under context Γ. Similarly, we can view $\theta_s(\cdot)$ as a function (in the ambient type theory, which is sythetic guarded domain theory) taking terms $\tilde{M}$ of type A avaiable *later* to terms of type A available *now*.

Notice that the notion of a "term available later" does not need to be part of the syntax of the intensional lambda calculus, because this can be expressed in the ambient theory. Similarly, we do not need a syntactic form to delay a term, because we can simply use next.

$$\text{Terms } M, N ::= \mho_A, \ \ldots\ \theta_s(\tilde{M})$$

$$\frac{\triangleright_t\ (\Gamma \vdash M_t : A)}{\Gamma \vdash \theta_s M : A}$$

$$\frac{\triangleright_t\ (\Gamma^{\sqsubseteq} \vdash M_t \sqsubseteq_i N_t : d)}{\Gamma^{\sqsubseteq} \vdash \theta_s M \sqsubseteq_i \theta_s N : d}$$

Recall that $\triangleright_t$ is a dependent form of $\triangleright$ where the arugment is allowed to mention t. In particular, here we apply the tick t to the later-terms M and N to get "now"-terms M_t and N_t.

Formally speaking, the term precision relation must be defined as a guarded fixpoint, i.e., we assume that the function is defined later, and use it to construct a definition that is defined "now". This involves applying a tick to the later-function to shift it to a now-function. Indeed, this is what we do in the formal Agda development, but in this paper, we will elide these issues as they are not relevant to the main ideas.

4 DOMAIN-THEORETIC CONSTRUCTIONS

In this section, we discuss the fundamental objects of the model into which we will embed the intensional lambda calculus and inequational theory. It is important to remember that the constructions in this section are entirely independent of the syntax described in the previous section; the notions defined here exist in their own right as purely mathematical constructs. In the next section, we will link the syntax and semantics via an interpretation function.

4.1 The Lift Monad

When thinking about how to model intensional gradually-typed programs, we must consider their possible behaviors. On the one hand, we have *failure*: a program may fail at run-time because of a type error. In addition to this, a program may "think", i.e., take a step of computation. If a program thinks forever, then it never returns a value, so we can think of the idea of thinking as a way of intensionally modelling *partiality*.

With this in mind, we can describe a semantic object that models these behaviors: a monad for embedding computations that has cases for failure and "thinking". Previous work has studied such a construct in the setting of the latter only, called the lift monad [**?**]; here, we add the additional effect of failure.

For a type A, we define the *lift monad with failure* $L_{\mho}A$, which we will just call the *lift monad*, as the following datatype:

$$\begin{aligned} L_{\mho}A := \\ &\eta : A \to L_{\mho}A \\ &\mho : L_{\mho}A \\ &\theta : \triangleright\ (L_{\mho}A) \to L_{\mho}A \end{aligned}$$

Formally, the lift monad $L_{\mho}A$ is defined as the solution to the guarded recursive type equation

$$L_{\mho}A \cong A + 1 + \triangleright\ L_{\mho}A.$$

An element of $L_{\mho}A$ should be viewed as a computation that can either (1) return a value (via η), (2) raise an error and stop (via $\mho$), or (3) think for a step (via θ).

Notice there is a computation fixθ of type $L_{\mathfrak{U}}A$. This represents a computation that thinks forever and never returns a value.

Since we claimed that $L_{\mathfrak{U}}A$ is a monad, we need to define the monadic operations and show that they repect the monadic laws. The return is just η, and extend is defined via by guarded recursion by cases on the input. Verifying that the monadic laws hold requires Löb-induction and is straightforward.

The lift monad has the following universal property. Let f be a function from A to B, where B is a $\triangleright$-algebra, i.e., there is $\theta_B : \triangleright B \to B$. Further suppose that B is also an "error-algebra", that is, an algebra of the constant functor $1 : \mathrm{Type} \to \mathrm{Type}$ mapping all types to Unit. This latter statement amounts to saying that there is a map Unit $\to B$, so B has a distinguished "error element" $\mathfrak{U}_B : B$.

Then there is a unique homomorphism of algebras $f' : L_{\mathfrak{U}}A \to B$ such that $f' \circ \eta = f$. The function $f'(l)$ is defined via guarded fixpoint by cases on l. In the $\mathfrak{U}$ case, we simply return $\mathfrak{U}_B$. In the $\theta(\tilde{l})$ case, we will return

$$\theta_B(\lambda t.(f'_t\ \tilde{l}_t)).$$

Recalling that f' is a guaded fixpoint, it is available "later" and by applying the tick we get a function we can apply "now"; for the argument, we apply the tick to $\tilde{l}$ to get a term of type $L_{\mathfrak{U}}A$.

4.1.1 Model-Theoretic Description. We can describe the lift monad in the topos of trees model as follows.

4.2 Predomains

The next important construction is that of a *predomain*. A predomain is intended to model the notion of error ordering that we want terms to have. Thus, we define a predomain A as a partially-ordered set, which consists of a type which we denote $\langle A \rangle$ and a reflexive, transitive, and antisymmetric relation $\leq_P$ on A.

For each type we want to represent, we define a predomain for the corresponding semantic type. For instance, we define a predomain for natural numbers, a predomain for the dynamic type, a predomain for functions, and a predomain for the lift monad. We describe each of these below.

We define monotone functions between predomain as expected. Given predomains A and B, we write $f : A \to_m B$ to indicate that f is a monotone function from A to B, i.e, for all $a_1 \leq_A a_2$, we have $f(a_1) \leq_B f(a_2)$.

- There is a predomain Nat for natural numbers, where the ordering is equality.
- There is a predomain Dyn to represent the dynamic type. The underlying type for this predomain is defined to be $\mathbb{N} + \triangleright\ (Dyn \to_m Dyn)$. This definition is valid because the occurrences of Dyn are guarded by the $\triangleright$. The ordering is defined via guarded recursion by cases on the argument, using the ordering on $\mathbb{N}$ and the ordering on monotone functions described below.
- For a predomain A, there is a predomain $L_{\mathfrak{U}}A$ that is the "lift" of A using the lift monad. We use the same notation for $L_{\mathfrak{U}}A$ when A is a type and A is a predomain, since the context should make clear which one we are referring to. The underling type of $L_{\mathfrak{U}}A$ is simply $L_{\mathfrak{U}}\langle A \rangle$, i.e., the lift of the underlying type of A. The ordering of $L_{\mathfrak{U}}A$ is the "lock-step error-ordering" which we describe in 4.3.
- For predomains A_i and A_o, we form the predomain of monotone functions from A_i to A_o, which we denote by $A_i \Rightarrow A_o$. Two such functions are related

Predomains will form the objects of a structure similar to a double category, with horizontal morphisms being monotone funcitons, and vertical morphisms being embedding-projection pairs discussed below.

4.3 Lock-step Error Ordering

As mentioned, the ordering on the lift of a predomain A The relation is parameterized by an ordering $\leq_A$ on A. We call this the lock-step error-ordering, the idea being that two computations l and l' are related if they are in lock-step with regard to their intensional behavior. That is, if l is ηx, then l' should be equal to ηy for some y such that $x \leq_A y$.

4.4 Weak Bisimilarity Relation

4.5 Combined Ordering

4.6 EP-Pairs

The use of embedding-projection pairs in gradual typing was introduced by New and Ahmed [**?**]. Here, we want to adapt this notion of embedding-projection pair to the setting of intensional denotaional semantics.

5 SEMANTICS

5.1 Types as Predomains

5.2 Terms as Monotone Functions

5.3 Type Precision as EP-Pairs

5.4 Term Precision via the Lock-Step Error Ordering

6 GRADUALITY

6.1 Outline

6.2 Extensional to Intensional

6.3 Intensional Results

6.4 Adequacy

6.5 Putting it Together

7 DISCUSSION

7.1 Synthetic Ordering

While the use of synthetic guarded domain theory allows us to very conveniently work with non-well-founded recursive constructions while abstracting away the precise details of step-indexing, we do work with the error ordering in a mostly analytic fashion in that gradual types are interpreted as sets equipped with an ordering relation, and all terms must be proven to be monotone. It is possible that a combination of synthetic guarded domain theory with *directed* type theory would allow for an a synthetic treatment of the error ordering as well.

REFERENCES

[1] Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 443–455. https://doi.org/10.1145/2837614.2837632

[2] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

[3] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274