

Denotational Semantics for Gradual Typing in Synthetic Guarded Domain Theory

ERIC GIOVANNINI and MAX S. NEW

We develop a denotational semantics for a simple gradually typed language 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, 15 pages. <https://doi.org/10.1145/1122445.1122456>

1 INTRODUCTION

1.1 Gradual Typing and Graduality

One of the principal categories on which type systems of programming languages are classified is that of static versus dynamic type discipline. In static typing, the code is type-checked at compile time, while in a dynamic typing, the type checking is deferred to run-time. Both approaches have benefits: with static typing, the programmer is assured that if the program passes the type-checker, their program is free of type errors. Meanwhile, dynamic typing allows the programmer to rapidly prototype their application code without needing to commit to fixed type signatures for their functions.

Gradually-typed languages [10] allow for both disciplines to be used in the same codebase, and support interoperability between statically-typed and dynamically-typed code. This flexibility allows programmers to begin their projects in a dynamic style and enjoy the benefits of dynamic typing related to rapid prototyping and easy modification while the codebase “solidifies”. Over time, as parts of the code become more mature and the programmer is more certain of what the types should be, the code can be *gradually* migrated to a statically typed style without needing to start the project over in a completely different language.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00

<https://doi.org/10.1145/1122445.1122456>

guarantees made by the static types. In particular, while statically-typed code can error at runtime in a gradually-typed language, such an error can always be traced back to a dynamically-typed term that violated the typing contract imposed by statically typed code. Second, gradually-typed languages 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 [11]. This property is also referred to as *graduality*, by analogy with parametricity. Intuitively, graduality says that 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 typing annotations will either result in the same behavior as the original, less precise program, or will result in a type error.

1.2 Current Approaches to Proving Graduality

Current approaches to proving graduality include the methods of Abstracting Gradual Typing [4] and the formal tools of the Gradualier [3]. These allow the language developer to start with a statically typed language and derive a gradually typed language that satisfies the gradual guarantee. The downside to these approaches is that the semantics of the resulting languages are too lazy: the frameworks consider only the β rules and not the η equalities. Furthermore, while these frameworks do prove graduality, they do not show the correctness of the equational theory, which is equally important to sound gradual typing. For example, programmers often refactor their code without thinking about whether the refactoring has broken the semantics of the program. It is the validity of the laws in the equational theory that guarantees that such refactorings are sound. Similarly, correctness of compiler optimizations rests on the validity of the corresponding equations from the equational theory. It is therefore important that the languages that claim to be gradually typed have provably correct equational theories.

New and Ahmed [8] have developed a semantic approach to specifying type dynamism in terms of *embedding-projection pairs*, which allows for a particularly elegant formulation of the gradual guarantee. Moreover, their axiomatic account of program equivalence allows for type-based reasoning about program equivalences. In this approach, a logical relation is constructed and shown to be sound with respect to the notion of observational approximation that specifies when one program is more precise than another. The downside of this approach is that each new language requires a different logical relation to prove graduality. Furthermore, the logical relations tend to be quite complicated due to a technical requirement known as *step-indexing*. As a result, developments using this approach tend to require vast effort, with the corresponding technical reports having 50+ pages of proofs.

An alternative approach, which we investigate in this paper, is provided by *synthetic guarded domain theory*. The techniques of synthetic guarded domain theory allow us to internalize the step-index reasoning normally required in logical relations proofs of graduality, ultimately allowing us to specify the logical relation in a manner that looks nearly identical to a typical, non-step-indexed logical relation.

In this paper, we report on work we have done to mechanize proofs of graduality and correctness of equational theories using SGDT techniques in Agda. Our goal in this work is to mechanize these proofs in a reusable way, thereby providing a framework to use to more easily and conveniently prove that existing languages satisfy graduality and have sound equational theories. Moreover, the aim is for designers of new languages to utilize the framework to facilitate the design of new provably-correct gradually-typed languages with nontrivial features.

1.3 Proving Graduality in SGGT

TODO: This section should probably be moved to after the relevant background has been introduced.

In this paper, we will utilize SGGT techniques to prove graduality for a particularly simple gradually-typed cast calculus, the gradually-typed lambda calculus. This is 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. The graduality theorem is shown below.

THEOREM 1.1 (GRADUALITY). *If $\cdot \vdash M \sqsubseteq N : \text{Nat}$, then*

- (1) $M \Downarrow \text{iff } N \Downarrow$
- (2) *If $M \Downarrow v_?$ and $N \Downarrow v'_?$ then either $v_? = \mathcal{U}$, or $v_? = v'_?$.*

Details can be found in later sections, but we provide a brief explanation of the terminology and notation:

- $M \sqsubseteq N : \text{Nat}$ means M and N are terms of type Nat such that M is “syntactically more precise” than N , or equivalently, N is “more dynamic” than M . Intuitively this means that M and N are the same except that in some places where M has explicit typing annotations, N has $?$ instead.
- $\cdot \Downarrow$ is a relation on terms that is defined such that $M \Downarrow$ means that M terminates, either with a run-time error or a value n of type Nat .
- \mathcal{U} is a syntactic representation of a run-time type error, which happens, for example, when a programmer tries to call a function with a value whose type is found to be incompatible with the argument type of the function.
- $v_?$ is shorthand for the syntactic representation of a term that is either equal to \mathcal{U} , or equal to the syntactic representation of a value n of type Nat .

Our first step toward proving graduality is to formulate an *step-sensitive*, or *intensional*, gradual lambda calculus, which we call $\text{Int-}\lambda$, in which the computation steps taken by a term are made explicit. The “normal” gradual lambda calculus for which we want to prove graduality will be called the *step-insensitive*, or *extensional*, gradual lambda calculus, denoted $\text{Ext-}\lambda$. We will define an erasure function $\lfloor \cdot \rfloor : \text{Int-}\lambda \rightarrow \text{Ext-}\lambda$ 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 $\text{Ext-}\lambda$ will have a corresponding program M_i in $\text{Int-}\lambda$ 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 $M_e \sqsubseteq_e M'_e : \text{Nat}$, we can find intensional programs M_i and M'_i that erase to them and are such that $M_i \sqsubseteq_i M'_i$. 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 .

1.4 Contributions

Our main contribution is a reusable framework in Guarded Cubical Agda for developing machine-checked proofs of graduality of a cast calculus. To demonstrate the feasibility 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.

1.5 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- λ) and the *intensional* gradual lambda calculus (Int- λ). 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 7, we outline in more detail the proof of graduality for the extensional gradual lambda calculus, which will make use of prove properties we prove about the intensional gradual lambda calculus. In Section 8, 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 Gradual Typing

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 *upcasted* to B , written $\langle B \leftarrow A \rangle M$, and a term N of type B may be *downcasted* to A , written $\langle A \leftarrow B \rangle N$. Upcasts always succeed, while downcasts may fail at runtime. We also have a notion of *syntactic 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.

2.2 Difficulties in Prior Semantics

In this work, we compare our approach to proving graduality to the approach introduced by New and Ahmed [8] which constructs a step-indexed logical relations model and shows that this model is sound with respect to their notion of contextual error approximation.

Because the dynamic type is modeled as a non-well-founded recursive type, their logical relation needs to be parameterized 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, because 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 transitivity.

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 require 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 .

2.3 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 [2].

SGDT offers a synthetic approach to domain theory that allows for guarded recursion to be expressed syntactically via a type constructor $\triangleright : \text{Type} \rightarrow \text{Type}$ (pronounced “later”). The use of a modality to express guarded recursion was introduced by Nakano [7]. 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 $\text{next} : A \rightarrow \triangleright A$ 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 *guarded fixpoint* operator

$$\text{fix} : \forall T, (\triangleright T \rightarrow T) \rightarrow T.$$

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 $\text{fix}f = f(\text{next}(\text{fix}f))$. In particular, this axiom applies to propositions $P : \text{Prop}$; proving a statement in this manner is known as Löb-induction.

The operators \triangleright , next , and fix described above can be indexed by objects called *clocks*. A clock serves as a reference relative to which steps are counted. 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 [1].

2.3.1 Ticked Cubical Type Theory. In Ticked Cubical Type Theory [?], there is an additional sort called *ticks*. Given a clock k , a tick $t : \text{tick}k$ 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 $\triangleright_t^k A$ 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 $t' : \text{tick}k$, 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 $\Gamma, t : \text{tick}k$ we have that M has type A , then in context Γ we have $\lambda t. M$ has type $\triangleright A$.

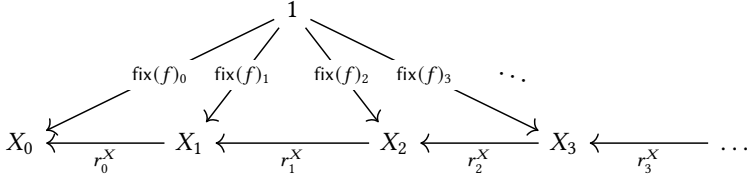
The statements in this paper have been formalized in a variant of Agda called Guarded Cubical Agda [?], an implementation of Clocked Cubical Type Theory.

$$X_0 \xleftarrow{r_0^X} X_1 \xleftarrow{r_1^X} X_2 \xleftarrow{r_2^X} X_3 \xleftarrow{r_3^X} \dots$$

Fig. 1. An example of an object in the topos of trees.

$$\begin{array}{ccccccc} X_0 & \xleftarrow{r_0^X} & X_1 & \xleftarrow{r_1^X} & X_2 & \xleftarrow{r_2^X} & X_3 \xleftarrow{r_3^X} \dots \\ f_0 \downarrow & & f_1 \downarrow & & f_2 \downarrow & & f_3 \downarrow \\ Y_0 & \xleftarrow{r_0^Y} & Y_1 & \xleftarrow{r_1^Y} & Y_2 & \xleftarrow{r_2^Y} & Y_3 \xleftarrow{r_3^Y} \dots \end{array}$$

Fig. 2. An example of a morphism in the topos of trees.

Fig. 3. The guarded fixpoint of f .

2.3.2 The Topos of Trees Model. The topos of trees model provides a useful intuition for reasoning in SGDT [2]. This section presupposes knowledge of category theory and can be safely skipped without disrupting the continuity.

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 $\hat{+}$). There is also an operator $\triangleright: \mathcal{U} \rightarrow \mathcal{U}$ such that $\text{El}(\triangleright(\text{next}A)) = \triangleright \text{El}(A)$, 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 $r_i^X: X_{i+1} \rightarrow X_i$ (see Figure 1). These should be thought of as “sets changing over time”, where X_i is the view of the set if we have $i + 1$ time steps to reason about it. We can also think of an ongoing computation, with X_i representing the potential results of the computation after it has run for $i + 1$ steps.

A morphism from $\{X_i\}$ to $\{Y_i\}$ is a family of functions $f_i: X_i \rightarrow Y_i$ that commute with the restriction maps in the obvious way, that is, $f_i \circ r_i^X = r_i^Y \circ f_{i+1}$ (see Figure 2).

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 $r_i^{\triangleright X} = !$, where $!$ is the unique map into 1, and $r_{i+1}^{\triangleright X} = r_i^X$. The morphism $\text{next}^X: X \rightarrow \triangleright X$ is defined pointwise by $\text{next}_0^X = !$, and $\text{next}_{i+1}^X = r_i^X$. It is easily checked that this satisfies the commutativity conditions required of a morphism in \mathcal{S} . Given a morphism $f: \triangleright X \rightarrow X$, i.e., a family of functions $f_i: (\triangleright X)_i \rightarrow X_i$ that commute with the restrictions in the appropriate way, we define $\text{fix}(f): 1 \rightarrow X$ pointwise by $\text{fix}(f)_i = f_i \circ \dots \circ f_0$. This can be visualized as a diagram in the category of sets as shown in Figure 3.

3 GTLC

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

Before diving into the details, let us give a brief overview of what we will define. We begin with a gradually-typed lambda calculus ($\text{Ext-}\lambda$), which is similar to the normal call-by-value gradually-typed lambda calculus, but differs in that it is actually a fragment of call-by-push-value specialized such that there are no non-trivial computation types. We do this for convenience, as either way we would need a distinction between values and effectful terms; the framework of call-by-push-value gives us a convenient language to define what we need.

We then show that composition of type precision derivations is admissible, as is heterogeneous transitivity for term precision, so it will suffice to consider a new language ($\text{Ext-}\lambda^{\text{trans}}$) in which we don't have composition of type precision derivations or heterogeneous transitivity of term precision.

We then observe that all casts, except those between Nat and $?$ and between $?$ and $?$, are admissible. This means it will be sufficient to consider a new language ($\text{Ext-}\lambda^{\text{trans-cast}}$) in which instead of having arbitrary casts, we have injections from Nat and $?$ into $?$, and case inspections from $?$ to Nat and $?$ to $?$.

From here, we define a *step-sensitive* (also called *intensional*) GSTLC, so-named because it makes the intensional stepping behavior of programs explicit in the syntax. This is accomplished by adding a syntactic “later” type and a syntactic θ that maps terms of type later A to terms of type A .

3.1 Syntax

The language is based on Call-By-Push-Value [5], and as such it has two kinds of types: *value types*, representing pure values, and *computation types*, representing potentially effectful computations. In the language, all computation types have the form $\text{Ret } A$ for some value type A . Given a value V of type A , the term $\text{ret } V$ views V as a term of computation type $\text{Ret } A$. Given a term M of computation type B , the term $\text{var } x = M \text{ in } N$ should be thought of as running M to a value V and then continuing as N , with V in place of x .

We also have value contexts and computation contexts, where the latter can be viewed as a pair consisting of (1) a stoup Σ , which is either empty or a hole of type B , and (2) a (potentially empty) value context Γ .

Value Types $A := \text{Nat} \mid ? \mid (A \multimap A')$
 Computation Types $B := \text{Ret } A$
 Value Contexts $\Gamma := \cdot \mid (\Gamma, x : A)$
 Computation Contexts $\Delta := \cdot \mid \bullet : B \mid \Delta, x : A$
 Values $V := \text{zro} \mid \text{suc } V \mid \langle B \multimap A \rangle V$
 Terms $M, N := \mathbf{U}_B \mid \text{ret } V \mid \text{var } x = M \text{ in } N \mid \lambda x. M \mid V_f V_x \mid \mid \langle A \multimap B \rangle M$

The value typing judgment is written $\Gamma \vdash V : A$ and the computation typing judgment is written $\Delta \vdash M : B$.

We define substitution for value contexts by the following rules:

$$\frac{\gamma : \Gamma' \rightarrow \Gamma \quad \Gamma' \vdash V : A}{(\gamma, V/x) : \Gamma' \rightarrow \Gamma, x : A} \quad \cdot : \cdot \rightarrow \cdot$$

We define substitution for computation contexts by the following rules:

$$\frac{\delta : \Delta' \rightarrow \Delta \quad \Delta'|_V \vdash V : A}{(\delta, V/x) : \Delta' \rightarrow \Delta, x : A} \quad \cdot : \cdot \rightarrow \cdot \quad \frac{\Delta' \vdash M : B}{M : \Delta' \rightarrow \bullet : B}$$

The typing rules are as expected, with a cast between A to B allowed only when $A \sqsubseteq B$. Notice that the upcast of a value is a value, since it always succeeds, while the downcast of a value is a computation, since it may fail.

$$\begin{array}{c} \frac{}{\cdot, \Gamma \vdash \mathbf{U}_B : B} \quad \frac{}{\Gamma \vdash \mathbf{zro} : \text{Nat}} \quad \frac{\Gamma \vdash V : \text{Nat}}{\Gamma \vdash \mathbf{suc } V : \text{Nat}} \quad \frac{\cdot, \Gamma, x : A \vdash M : \text{Ret } A'}{\Gamma \vdash \lambda x. M : A \rightarrow A'} \\[10pt] \frac{\Gamma \vdash V_f : A \rightarrow A' \quad \Gamma \vdash V_x : A}{\cdot, \Gamma \vdash V_f V_x : \text{Ret } A'} \quad \frac{\Gamma \vdash V : A}{\cdot, \Gamma \vdash \mathbf{ret } V : \text{Ret } A} \\[10pt] \frac{\Delta \vdash M : \text{Ret } A \quad \cdot, \Delta|_V, x : A \vdash N : B}{\Delta \vdash \mathbf{var } x = M \text{ in } N : B} \quad \frac{A \sqsubseteq A' \quad \Gamma \vdash V : A}{\Gamma \vdash \langle A' \leftarrow V \rangle : A'} \quad \frac{A \sqsubseteq A' \quad \Gamma \vdash V : A'}{\cdot, \Gamma \vdash \langle A \leftarrow A' \rangle V : \text{Ret } A} \end{array}$$

In the equational theory, we have β and η laws for function type, as well a β and η law for $\text{Ret } A$.

$$\begin{array}{c} \frac{\cdot, \Gamma, x : A \vdash M : \text{Ret } A' \quad \Gamma \vdash V : A}{(\lambda x. M) V = M[V/x]} \quad \frac{\Gamma \vdash V : A \rightarrow A}{\Gamma \vdash V = \lambda x. V x} \quad \mathbf{var } x = \mathbf{ret } V \text{ in } N = N[V/x] \\[10pt] \frac{\bullet : \text{Ret } A, \Gamma \vdash M : B}{\bullet : \text{Ret } A, \Gamma \vdash M = \mathbf{var } x = \bullet \text{ in } M[\mathbf{ret } x]} \end{array}$$

3.2 Type Precision

The type precision rules specify what it means for a type A to be more precise than A' . We have reflexivity rules for $?$ and Nat , as well as rules that Nat is more precise than $?$ and $?\rightarrow?$ is more precise than $?$. We also have a transitivity rule for composition of type precision, and also a rule for function types stating that given $A_i \sqsubseteq A'_i$ and $A_o \sqsubseteq A'_o$, we can prove $A_i \rightarrow A_o \sqsubseteq A'_i \rightarrow A'_o$. Finally, we can lift a relation on value types $A \sqsubseteq A'$ to a relation $\text{Ret } A \sqsubseteq \text{Ret } A'$ on computation types.

$$\begin{array}{c} \frac{}{? \sqsubseteq ?} \quad \frac{}{\text{Nat} \sqsubseteq \text{Nat}} \text{Nat} \quad \frac{}{\text{Nat} \sqsubseteq ?} \text{Inj}_{\text{Nat}} \quad \frac{A_i \sqsubseteq A'_i \quad A_o \sqsubseteq A'_o}{(A_i \Rightarrow A_o) \sqsubseteq (A'_i \Rightarrow A'_o)} \Rightarrow \\[10pt] \frac{}{(? \rightarrow ?) \sqsubseteq ?} \text{Inj}_{\Rightarrow} \quad \frac{A \sqsubseteq A' \quad A' \sqsubseteq A''}{A \sqsubseteq A''} \text{ValComp} \quad \frac{B \sqsubseteq B' \quad B' \sqsubseteq B''}{B \sqsubseteq B''} \text{CompComp} \\[10pt] \frac{A \sqsubseteq A'}{\text{Ret } A \sqsubseteq \text{Ret } A'} \text{Ret} \end{array}$$

Note that as a consequence of this presentation of the type precision rules, we have that if $A \sqsubseteq A'$, 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 $A \sqsubseteq A'$, we have a derivation $c : A \sqsubseteq A'$ 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 $c_i : A_i \sqsubseteq A_i$ and $c_o : A_o \sqsubseteq A'_o$, then there is a derivation $c_i \Rightarrow c_o : (A_i \Rightarrow A_o) \sqsubseteq (A'_i \Rightarrow A'_o)$. Likewise for the remaining 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 $c : A \sqsubseteq A'$ and $c' : A' \sqsubseteq A''$, we can define (via the rule ValComp) their composition $c \odot c' : A \sqsubseteq A''$. The same holds for computation type precision derivations. This notion will be used below in the statement of transitivity of the term precision relation.

3.3 Term Precision

We allow for a *heterogeneous* term precision judgment on terms values V of type A and V' of type A' provided that $A \sqsubseteq A'$ holds. Likewise, for computation types $B \sqsubseteq B'$, if M has type B and M' has type B' , we can form the judgment that $M \sqsubseteq M'$.

In order to deal with open terms, we will need the notion of a type precision *context*, which we denote Γ^\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. Similarly, we have computation type precision contexts Δ^\sqsubseteq . Similar to “normal” computation type precision contexts Δ , these consist of (1) a stoup Σ which is either empty or has a hole $\bullet : d$ for some computation type precision derivation d , and (2) a value type precision context Γ^\sqsubseteq .

As with type precision derivations, we write Γ to mean the context of reflexivity derivations $\Gamma(x) \sqsubseteq \Gamma(x)$. Likewise for computation type precision contexts. Furthermore, we write $\Gamma_1^\sqsubseteq \odot \Gamma_2^\sqsubseteq$ to denote the “composition” of Γ_1^\sqsubseteq and Γ_2^\sqsubseteq — that is, the precision context whose value at x is the type precision derivation $\Gamma_1^\sqsubseteq(x) \odot \Gamma_2^\sqsubseteq(x)$. This of course assumes that each of the type precision derivations is composable, i.e., that the RHS of $\Gamma_1^\sqsubseteq(x)$ is the same as the left-hand side of $\Gamma_2^\sqsubseteq(x)$. We define the same for computation type precision contexts Δ_1^\sqsubseteq and Δ_2^\sqsubseteq , provided that both the computation type precision contexts have the same “shape”, which is defined as (1) either the stoup is empty in both, or the stoup has a hole in both, say $\bullet : d$ and $\bullet : d'$ where d and d' are composable, and (2) their value type precision contexts are composable as described above.

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:

$$\begin{array}{c}
\frac{c : A \sqsubseteq B \quad \Gamma^\sqsubseteq(x) = (A, B)}{\Gamma^\sqsubseteq \vdash x \sqsubseteq_e x : c} \text{VAR} \qquad \frac{}{\Gamma^\sqsubseteq \vdash \text{zro} \sqsubseteq_e \text{zro} : \text{Nat}} \text{ZRO} \\
\\
\frac{\Gamma^\sqsubseteq \vdash V \sqsubseteq_e V' : \text{Nat}}{\Gamma^\sqsubseteq \vdash \text{suc } V \sqsubseteq_e \text{suc } V' : \text{Nat}} \text{SUC} \\
\\
\frac{c_i : A_i \sqsubseteq A'_i \quad c_o : A_o \sqsubseteq A'_o \quad , \Gamma^\sqsubseteq, x : c_i \vdash M \sqsubseteq_e M' : \text{Ret } c_o}{\Gamma^\sqsubseteq \vdash \lambda x. M \sqsubseteq_e \lambda x. M' : (c_i \rightarrow c_o)} \text{LAMBDA} \\
\\
\frac{\Gamma^\sqsubseteq \vdash V_f \sqsubseteq_e V'_f : (c_i \rightarrow c_o) \quad \Gamma^\sqsubseteq \vdash V_x \sqsubseteq_e V'_x : c_i}{, \Gamma^\sqsubseteq \vdash V_f V_x \sqsubseteq_e V'_f V'_x : \text{Ret } c_o} \text{APP} \qquad \frac{\Gamma^\sqsubseteq \vdash V \sqsubseteq_e V' : c}{, \Gamma^\sqsubseteq \vdash \text{ret } V \sqsubseteq_e \text{ret } V' : \text{Ret } c} \text{RET} \\
\\
\frac{\Delta^\sqsubseteq \vdash M \sqsubseteq_e M' : \text{Ret } c \quad , \Delta^\sqsubseteq|_V, x : c \vdash N \sqsubseteq_e N' : d}{\Delta^\sqsubseteq \vdash \text{var } x = M \text{ in } N \sqsubseteq_e \text{var } x = M' \text{ in } N' : d} \text{BIND}
\end{array}$$

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

$$\begin{array}{c}
\frac{\Delta_1^\sqsubseteq \vdash M : B}{\Delta \vdash \mathcal{U}_B \sqsubseteq_e M : B} \mathcal{U} \qquad \frac{\Delta_1^\sqsubseteq \vdash M \sqsubseteq_e M' : d \quad \Delta_2^\sqsubseteq \vdash M' \sqsubseteq_e M'' : d'}{\Delta_1^\sqsubseteq \odot \Delta_2^\sqsubseteq \vdash M \sqsubseteq_e M'' : d \odot d'} \text{TRANSITIVITY} \\
\\
\frac{\Gamma, x : A_i \vdash M : A_o \quad \Gamma \vdash V : A_i}{\Gamma^\sqsubseteq \vdash (\lambda x. M) V \sqsupseteq_e M[V/x] : A_o} \beta\text{-FUN} \qquad \frac{\Gamma \vdash V : A_i \Rightarrow A_o}{\Gamma^\sqsubseteq \vdash \lambda x. (V x) \sqsupseteq_e V : A_i \Rightarrow A_o} \eta\text{-FUN} \\
\\
\text{var } x = \text{ret } V \text{ in } N = N[V/x] \quad \beta\text{-RET} \qquad \frac{\bullet : \text{Ret } A, \Gamma \vdash M : B}{\bullet : \text{Ret } A, \Gamma \vdash M = \text{var } x = \bullet \text{ in } M[\text{ret } x]} \eta\text{-RET} \\
\\
\frac{d : A \sqsubseteq B \quad \Gamma \vdash M : A}{\Gamma^\sqsubseteq \vdash M \sqsubseteq_e \langle B \prec_\prec A \rangle M : d} \text{UPR} \qquad \frac{d : A \sqsubseteq B \quad \Gamma^\sqsubseteq \vdash M \sqsubseteq_e N : d}{\Gamma^\sqsubseteq \vdash \langle B \prec_\prec A \rangle M \sqsubseteq_e N : B} \text{UPL} \\
\\
\frac{d : A \sqsubseteq B \quad \Gamma \vdash M : B}{\Gamma^\sqsubseteq \vdash \langle A \prec_\prec B \rangle M \sqsubseteq_e M : d} \text{DnL} \qquad \frac{d : A \sqsubseteq B \quad \Gamma^\sqsubseteq \vdash M \sqsubseteq_e N : d}{\Gamma^\sqsubseteq \vdash M \sqsubseteq_e \langle A \prec_\prec B \rangle N : A} \text{DnR}
\end{array}$$

The rules UpR, UpL, DnL, and DnR were introduced in [9] 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.

$$\begin{array}{ccc}
 \Gamma & \xrightarrow{M} & A \\
 \Downarrow & & \Downarrow \\
 \Gamma' & \xrightarrow{M'} & A' \\
 \Downarrow & & \Downarrow \\
 \Gamma'' & \xrightarrow{M''} & A''
 \end{array}$$

Fig. 4. Heterogeneous transitivity.

3.4 Removing Transitivity

The first observation we make is that transitivity of type precision, and heterogeneous transitivity of term precision, are admissible. That is, consider a related language which is the same as Ext- λ except that we have removed the composition rule for type precision and the heterogeneous transitivity rule for type precision. Denote this language by Ext- λ^{trans} . We claim that in this new language, the rules we removed are derivable from the remaining rules. More specifically, consider the following situation in Ext- λ :

TODO

3.5 Removing Casts

We now observe that all casts, except those between Nat and ? and between ? \rightarrow ? and ?, are admissible. Consider a new language (Ext- $\lambda^{\text{trans-cast}}$) in which instead of having arbitrary casts, we have injections from Nat and ? \rightarrow ? into ?, and case inspections from ? to Nat and ? to ? \rightarrow ?. We claim that in Ext- $\lambda^{\text{trans-cast}}$, all of the casts present in Ext- λ^{trans} are derivable. It will suffice to verify that casts for function type are derivable. This holds because function casts are constructed inductively from the cast for their domain and codomain. The base case is one of the casts involving Nat or ? \rightarrow ? which are present in Ext- $\lambda^{\text{trans-cast}}$ as injections and case inspections.

The resulting calculus now lacks transitivity of type precision, heterogeneous transitivity of term precision, and arbitrary casts. In this setting, rather than type precision, it makes more sense to speak of arbitrary monotone relations on types, which we denote by $A \multimap A'$. We have relations on value types, as well as on computation types.

Value Relations $R := \text{Nat} \mid ? \mid (R \multimap R) \mid ?$

Computation Relations $S := \text{Lift}R$

Value Relation Contexts $\Gamma^{\bullet\bullet} := \cdot \mid \Gamma^{\bullet\bullet}, A^{\bullet\bullet}(x_l : A_l, x_r : A_r)$

Computation Relation Contexts $\Delta^{\bullet\bullet} := \cdot \mid \bullet : B^{\bullet\bullet} \mid \Delta^{\bullet\bullet}, A^{\bullet\bullet}(x_l : A_l, x_r : A_r)$

3.6 The Step-Sensitive Lambda Calculus

From here, we define an *step-sensitive* (also called *intensional*) GSTLC. As mentioned, this language makes the intensional stepping behavior of programs explicit in the syntax. We do this by adding a syntactic “later” type and a syntactic θ that maps terms of type later A to terms of type A .

In the step-sensitive syntax, we add a type constructor for later, as well as a syntactic θ term and a syntactic next term. We add rules for each of these, and also modify the rules for inj-arr and case-arr, since now the function is not $\text{Dyn} \rightarrow \text{Dyn}$ but rather $\triangleright (\text{Dyn} \rightarrow \text{Dyn})$.

We define an erasure function from step-sensitive syntax to step-insensitive syntax by induction on the step-sensitive types and terms. The basic idea is that the syntactic type $\triangleright A$ erases to A , and next and θ erase to the identity.

3.7 Quotienting by Syntactic Bisimilarity

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, called the lift monad [6]; here, we augment it with the additional effect of failure.

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

$$\begin{aligned} L_{\mathcal{U}}A &:= \\ \eta &: A \rightarrow L_{\mathcal{U}}A \\ \mathcal{U} &: L_{\mathcal{U}}A \\ \theta &: \triangleright (L_{\mathcal{U}}A) \rightarrow L_{\mathcal{U}}A \end{aligned}$$

Unless otherwise mentioned, all constructs involving \triangleright or fix are understood to be with respect to a fixed clock k . So for the above, we really have for each clock k a type $L_{\mathcal{U}}^k A$ with respect to that clock.

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

$$L_{\mathcal{U}}A \cong A + 1 + \triangleright L_{\mathcal{U}}A.$$

An element of $L_{\mathcal{U}}A$ should be viewed as a computation that can either (1) return a value (via η), (2) raise an error and stop (via \mathcal{U}), or (3) think for a step (via θ). Notice there is a computation $\text{fix}\theta$ of type $L_{\mathcal{U}}A$. This represents a computation that thinks forever and never returns a value.

Since we claimed that $L_{\mathcal{U}}A$ is a monad, we need to define the monadic operations and show that they respect the monadic laws. The return is just η , and extend is defined via 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 $\theta_B: \triangleright B \rightarrow B$. Further suppose that B is also an “error-algebra”, that is, an algebra of the constant functor $1: \text{Type} \rightarrow \text{Type}$ mapping all types to Unit . This latter statement amounts to saying that there is a map $\text{Unit} \rightarrow B$, so B has a distinguished “error element” $\mathcal{U}_B: B$.

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

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 $f: A \rightarrow_m B$ 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 by guarded fixpoint to be such that $\langle \text{Dyn} \rangle \cong \mathbb{N} + \triangleright (\langle \text{Dyn} \rangle \rightarrow_m \langle \text{Dyn} \rangle)$. 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 $L_{\mathcal{U}}A$ for the “lift” of A using the lift monad. We use the same notation for $L_{\mathcal{U}}A$ when A is a type and A is a predomain, since the context should make clear which one we are referring to. The underlying type of $L_{\mathcal{U}}A$ is simply $L_{\mathcal{U}}\langle A \rangle$, i.e., the lift of the underlying type of A . The ordering on $L_{\mathcal{U}}A$ is the “step-sensitive 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 $A_i \Rightarrow A_o$. The ordering is such that f is below g if for all a in $\langle A_i \rangle$, we have $f(a)$ is below $g(a)$ in the ordering for A_o .

4.3 Step-Sensitive Error Ordering

As mentioned, the ordering on the lift of a predomain A is called the *step-sensitive error-ordering* (also called “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, up to l erroring. Formally, we define this ordering as follows:

- $\eta x \lesssim \eta y$ if $x \leq_A y$.
- $\mathcal{U} \lesssim l$ for all l
- $\theta \tilde{r} \lesssim \theta \tilde{r}'$ if $\triangleright_t (\tilde{r}_t \lesssim \tilde{r}'_t)$

We also define a heterogeneous version of this ordering between the lifts of two different predomains A and B , parameterized by a relation R between A and B .

4.4 Step-Insensitive Relation

We define another ordering on $L_{\mathcal{U}}A$, called the “step-insensitive ordering” or “weak bisimilarity”, written $l \approx l'$. Intuitively, we say $l \approx l'$ if they are equivalent “up to delays”. We introduce the notation $x \sim_A y$ to mean $x \leq_A y$ and $y \leq_A x$.

The weak bisimilarity relation is defined by guarded fixpoint as follows:

$$\begin{aligned}
 &\mathcal{U} \approx \mathcal{U} \\
 &\eta x \approx \eta y \text{ if } x \sim_A y \\
 &\theta \tilde{x} \approx \theta \tilde{y} \text{ if } \triangleright_t (\tilde{x}_t \approx \tilde{y}_t) \\
 &\theta \tilde{x} \approx \mathcal{U} \text{ if } \theta \tilde{x} = \delta^n(\mathcal{U}) \text{ for some } n \\
 &\theta \tilde{x} \approx \eta y \text{ if } (\theta \tilde{x} = \delta^n(\eta x)) \text{ for some } n \text{ and } x : \langle A \rangle \text{ such that } x \sim_A y \\
 &\mathcal{U} \approx \theta \tilde{y} \text{ if } \theta \tilde{y} = \delta^n(\mathcal{U}) \text{ for some } n \\
 &\eta x \approx \theta \tilde{y} \text{ if } (\theta \tilde{y} = \delta^n(\eta y)) \text{ for some } n \text{ and } y : \langle A \rangle \text{ such that } x \sim_A y
 \end{aligned}$$

4.5 Error Domains

4.6 Globalization

Recall that in the above definitions, any occurrences of \triangleright were with respect to a fixed clock k . Intuitively, this corresponds to a step-indexed set. It will be necessary to consider the “globalization” of these definitions, i.e., the “global” behavior of the type over all potential time steps. This is accomplished in the type theory by *clock quantification* [1], whereby given a type X parameterized by a clock k , we consider the type $\forall k. X[k]$. This corresponds to leaving the step-indexed world and passing to the usual semantics in the category of sets.

5 SEMANTICS

5.1 Relational Semantics

5.1.1 Term Precision via the Step-Sensitive Error Ordering.

6 UNARY CANONICITY

Before discussing graduality, we seek to prove its “unary” analogue. Namely, instead of considering inequality between terms, we start by considering equality.

7 GRADUALITY

The main theorem we would like to prove is the following:

THEOREM 7.1 (GRADUALITY). *If $\cdot \vdash M \sqsubseteq N : \text{Nat}$, then*

- (1) *If $N = \mathcal{U}$, then $M = \mathcal{U}$*
- (2) *If $N = 'n$, then $M = \mathcal{U}$ or $M = 'n$*
- (3) *If $M = V$, then $N = V$*

8 DISCUSSION

REFERENCES

- [1] Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) (ICFP '13). Association for Computing Machinery, New York, NY, USA, 197–208. <https://doi.org/10.1145/2500365.2500597>

- [2] Lars Birkedal, Rasmus Ejlers Mogelberg, Jan Schwinghammer, and Kristian Stovring. 2011. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. 55–64. <https://doi.org/10.1109/LICS.2011.16>
- [3] 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>
- [4] 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>
- [5] Paul Blain Levy. 2001. *Call-by-Push-Value*. Ph. D. Dissertation. Queen Mary, University of London, London, UK.
- [6] Rasmus Ejlers Møgelberg and Marco Paviotti. 2016. Denotational Semantics of Recursive Types in Synthetic Guarded Domain Theory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) (LICS '16). Association for Computing Machinery, New York, NY, USA, 317–326. <https://doi.org/10.1145/2933575.2934516>
- [7] H. Nakano. 2000. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. 255–266. <https://doi.org/10.1109/LICS.2000.855774>
- [8] Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proc. ACM Program. Lang.* 2, ICFP, Article 73 (jul 2018), 30 pages. <https://doi.org/10.1145/3236768>
- [9] Max S. New and Daniel R. Licata. 2018. Call-by-name Gradual Type Theory. *CoRR* (2018). <http://arxiv.org/abs/1802.00061>
- [10] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. 81–92.
- [11] 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>