Call-by-Name Gradual Type Theory

Max S. New[1] and Daniel R. Licata[2]

1 **Northeastern University, Boston, USA**
`maxnew@ccs.neu.edu`
2 **Wesleyan University, Middletown, USA**
`dlicata@wesleyan.edu`

Abstract

We present *gradual type theory*, a logic and type theory for call-by-name gradual typing. We define the central constructions of gradual typing (the dynamic type, type casts and type error) in a novel way, by universal properties relative to new judgments for *gradual type and term dynamism*, which were developed in blame calculi and to state the "gradual guarantee" theorem of gradual typing. Combined with the ordinary extensionality (η) principles that type theory provides, we show that most of the standard operational behavior of casts is *uniquely determined* by the gradual guarantee. This provides a semantic justification for the definitions of casts, and shows that non-standard definitions of casts must violate these principles. Our type theory is the internal language of a certain class of preorder categories called *equipments*. We give a general construction of an equipment interpreting gradual type theory from a 2-category representing non-gradual types and programs, which is a semantic analogue of Findler and Felleisen's definitions of contracts, and use it to build some concrete domain-theoretic models of gradual typing.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Gradually typed languages allow for static and dynamic programming styles within the same language. They are designed with twin goals of allowing easy interoperability between static and dynamic portions of a codebase and facilitating a smooth transition from dynamic to static typing. This allows for the introduction of new typing features to legacy languages and codebases without the enormous manual effort currently necessary to migrate code from a dynamically typed language to a fully statically typed language. Gradual typing allows exploratory programming and prototyping to be done in a forgiving, dynamically typed style, while later that code can be typed to ease readability and refactoring. Due to this appeal, there has been a great deal of research on extending gradual typing [28, 24] to numerous language features such as parametric polymorphism [1, 13], effect tracking [2], typestate [32], session types [12], and refinement types [15]. Almost all work on gradual typing is based solely on operational semantics, and recent work such as [23] has codified some of the central design principles of gradual typing in an operational setting. In this paper, we are interested in complementing this operational work with a type-theoretic and category-theoretic analysis of these design principles. We believe this will improve our understanding of gradually typed languages, particularly with respect to principles for reasoning about program equivalence, and assist in designing and evaluating new gradually typed languages.

One of the central design principles for gradual typing is *gradual type soundness*. At its most general, this should mean that the types of the gradually typed language provide the same type-based reasoning that one could reasonably expect from a similar statically typed language, i.e. one with effects. While this has previously been defined using operational semantics and a notion of *blame* [30], the idea of soundness we consider here is that the types should provide the same extensionality (η) principles as in a statically typed language.

This way, programmers can reason about the "typed" parts of gradual programs in the same way as in a fully static language. This definition fits nicely with a category-theoretic perspective, because the β and η principles correspond to definitions of connectives by a *universal property*. The second design principle is the *gradual guarantee* [23], which we will refer to as *graduality* (by analogy with parametricity). Informally, graduality of a language means that syntactic changes from dynamic to static typing (or vice-versa) should result in simple, predictable changes to the semantics of a term. More specifically, if a portion of a program is made "more static"/"less dynamic" then the new program should either have the same behavior or result in a runtime type error. Other observable behavior such as values produced, I/O actions performed or termination should not be changed.

In this paper, we codify these two principles of soundness and graduality *directly* into a logical syntax we dub (call-by-name) *Gradual Type Theory* (Section 2). For graduality, we develop a logic of *type and term dynamism* that can be used to reason about the relationship between "more dynamic" and "less dynamic" versions of a program, and to give a novel *uniform specification* (universal property) for the dynamic type, type errors, and the runtime type casts of a gradually typed language. For soundness, we assert β and η principles as axioms of term dynamism, so it can also be used to reason about programs' behavior. Furthermore, using the η principles for types, we show that most of the operational rules of runtime casts of existing (call-by-name) gradually typed languages are *uniquely determined* by these constraints of soundness and graduality (Section 3). For example, uniqueness implies that any enforcement scheme in a specific gradually typed language that is *not* equivalent to the standard "wrapping" ones *must* violate either soundness or graduality. We have chosen call-by-name because it is a simple setting with the necessary η principles (for negative types) to illustrate our technique; we leave call-by-value gradual type theory to future work.

We give a sound and complete category theoretic semantics for gradual type theory in terms of certain *preorder categories* (double categories where one direction is thin) (Section 4). We show that the contract interpretation of gradual typing [29] can be understood as a tool for constructing models (Section 5): starting from some existing language/category C, we first implement casts as suitable pairs of functions/morphisms from C, and then equip every type with canonical casts to the dynamic type. We apply this to construct some concrete models in domains (Section 6). Conceptually, gradual type theory is analogous to Moggi's *monadic metalanguage* [18]: it clarifies general principles present in many different programming languages; it is the internal language of a quite general class of category-theoretic structures; and, for a specific language, a number of useful results can be proved all at once by showing that a logical relation over it is a model of the type theory.

A logic of dynamism and casts Before proceeding to the technical details, we explain at a high level how our type theory accounts for two key features of gradual typing: graduality and casts. The "gradual guarantee" as defined in [23] applies to a surface language where runtime type casts are implicitly inserted based on type annotations, but we will focus here on an analysis of fully elaborated languages, where explicit casts have already been inserted (so our work does not yet address gradual type checking). The gradual guarantee as defined in [23] makes use of a *syntactically less dynamic* ordering on types: the dynamic type (universal domain) ? is the most dynamic, and A is less dynamic than B if B has the same structure as A but some sub-terms are replaced with ? (for example, $A \to (B \times C)$ is less dynamic than $? \to (B \times ?)$, $? \to ?$ and ?). Intuitively, a less dynamic type constrains the behavior of the program more, but consequently gives stronger reasoning principles. This notion is extended to closed well-typed *terms* $t : A$ and $t' : A'$ with A less dynamic than A': t is *syntactically less dynamic* than t' if t is obtained from t' by replacing the input and

output type of each type cast with a less (or equally) dynamic type (in [23] this was called "precision"). For example, if $add1 : ? \to \mathbb{N}$ and $\mathsf{true} : ?$, then $add1((? \Leftarrow \mathbb{N})(\mathbb{N} \Leftarrow ?)\mathsf{true})$ (cast true from dynamic to $\mathbb{N}$ and back, to assert it is a number) is syntactically less dynamic than $add1((? \Leftarrow ?)(? \Leftarrow ?)\mathsf{true})$ (where both casts are the identity). Then the gradual guarantee [23] says that if t is syntactically less dynamic than t', then t is *semantically less dynamic* than t': either t evaluates to a type error (in which case t' may do anything) or t, t' have the same first-order behavior (both diverge or both terminate with t producing a less dynamic value). In the above example, the less dynamic term always errors (because true fails the runtime $\mathbb{N}$ check), while the more dynamic term only errors if $add1$ uses its argument as a number. In contrast, a program that returns a different value than $add1(\mathsf{true})$ does will not be semantically less dynamic than it.

The approach we take in this paper is to give a *syntactic logic* for the *semantic* notion of one term being less dynamic than another, with $\mathcal{U}$ (type error) the least element, and all term constructors monotone. We call this the *term dynamism relation* $t \sqsubseteq t'$, and it includes not only syntactic changes in type casts, as above, but also equational laws like identity and composition for casts, and $\beta\eta$ rules—so $t \sqsubseteq t'$ intuitively means that t type-errors more than (or as much as) t', but is otherwise equal according to these equational laws. A programming language that is a model of our type theory will therefore be equipped with a semantic $t[\![\sqsubseteq]\!]t'$ relation validating these rules, so $t[\![\sqsubseteq]\!]t'$ if t type-errors more than t' up to these equational and monotonicity laws. In particular, making type cast annotations less dynamic will result in related programs, and if $[\![\sqsubseteq]\!]$ is adequate (doesn't equate operationally distinguishable terms), then this implies the gradual guarantee [23]. Therefore, we say a model "satisfies graduality" in the same sense that a language satisfies parametricity.

Next, we discuss the relationship between term dynamism and casts/contracts, one of the most novel parts of our theory. Explicit casts in a gradually typed language are typically presented by the syntactic form $(B \Leftarrow A)t$, and their semantics is either defined by various operational reductions that inspect the structure of A and B, or by "contract" translations, which compile a language with casts to another language, where the casts are implemented as ordinary functions . In both cases, the behavior of casts is defined by inspection on types and part of the language definition, with little justification beyond intuition and precedent.

In gradual type theory, on the other hand, the behavior of casts is *not* defined by inspection of types. Rather, we use the new type and term dynamism judgments, which are defined *prior to* casts, to give a few simple and uniform rules specifying casts in all types via a universal property (optimal implementation of a specification). Our methodology requires isolating two special subclasses of casts, upcasts and downcasts. An upcast goes from a "more static" to a "more dynamic" type— for instance $(? \Leftarrow (A \to B))$ is an upcast from a function type up to the dynamic type—whereas a downcast is the opposite, casting to the more static type. We represent the relationship "A is less dynamic than B" by a *type dynamism* judgment $A \sqsubseteq B$ (which corresponds to the "naïve subtyping" of [30]). In gradual type theory, the upcast $\langle B \leftarrowtail A \rangle$ from A to B and the downcast $\langle A \twoheadleftarrow B \rangle$ from B to A can be formed whenever $A \sqsubseteq B$. This leaves out certain casts like $(? \times \mathbb{N}) \Leftarrow (\mathbb{N} \times ?)$ where neither type is more dynamic than the other. However, as first recognized in [11], these casts are macro-expressible [7] as a composite of an upcast to the dynamic type and then a downcast from it (define $(B \Leftarrow A)t$ as the composite $\langle B \twoheadleftarrow ? \rangle \langle ? \leftarrowtail A \rangle t$).

A key insight is that we can give upcasts and downcasts dual specifications using term dynamism, which say how the casts relate programs to type dynamism. If $A \sqsubseteq B$, then for any term $t : A$, the upcast $\langle B \leftarrowtail A \rangle t : B$ is the *least* dynamic term of type B that is more dynamic than t. In order-theoretic terms, $\langle B \leftarrowtail A \rangle t : B$ is the $\sqsubseteq$-meet of all terms $u : B$ with

$$\frac{A\,\text{type} \qquad A'\,\text{type}}{A \sqsubseteq A'} \qquad \frac{\Gamma\,\text{context} \qquad \Gamma'\,\text{context}}{\Phi : \Gamma \sqsubseteq \Gamma'} \qquad \frac{\Gamma\,\text{context} \qquad A\,\text{type}}{\Gamma \vdash t : A} \qquad \frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Gamma \vdash t : A \qquad A \sqsubseteq A' \qquad \Gamma' \vdash t' : A'}{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'}$$

Figure 1 Judgment Presuppositions of Preorder Type Theory

$$\frac{X\ \text{base type}}{X\,\text{type}} \qquad \frac{}{\cdot\,\text{context}} \qquad \frac{\Gamma\,\text{context} \qquad A\,\text{type} \qquad x \notin \text{dom}(\Gamma)}{\Gamma, x : A\,\text{context}} \qquad \frac{}{\Gamma, x : A, \Gamma' \vdash x : A}$$

Figure 2 Preorder Type Theory: Type and Term Structure

$t \sqsubseteq u$. Downcasts have a dual interpretation as a $\sqsubseteq$-join. Intuitively, this property means upcast $\langle B \leftarrow A \rangle t$ behaves as much as possible like t itself, while supporting the additional interface provided by expanding the type from A to B.

This simple definition has powerful consequences that we explore in Section 3, because it characterizes the upcasts and downcasts up to program equivalence. We show that standard implementations of casts are the *unique* implementations that satisfy β, η and and basic congruence rules. In fact, almost all of the standard operational rules of a simple call-by-name gradually typed language are term-dynamism equivalences in gradual type theory. The exception is rules that rely on disjointness of different type connectives (such as $\langle ? \to ? \twoheadleftarrow ? \rangle \langle ? \leftarrow ? \times ? \rangle t \mapsto \Omega$), which are independent, and can be added as axioms.

2 Gradual Type Theory

In this section, we present the rules of gradual type theory (GTT). Gradual type theory presents the types, connectives and casts of gradual typing in a modular, type-theoretic way: the dynamic type and casts are defined by rules using the *judgmental structure* of the type theory, which extends the usual judgmental structure of call-by-name typed lambda calculus with a syntax for type and term dynamism. Since the judgmental structure is as important as these types, we present a bare *preorder type theory* (PTT) with no types first. Then we can modularly define what it means for this theory to have a dynamic type, casts, functions and products, and gradual type theory is preorder type theory with all of these.

Preorder Type Theory Preorder type theory (PTT) has 6 judgments: types, contexts, type dynamism, dynamism contexts, terms and term dynamism. Their presuppositions (one is only allowed to make a judgment when these conditions hold) are presented in Figure 1, where A type and Γ context have no conditions. The types, contexts and terms (Figure 2) are structured as a standard call-by-name type theory. Terms are treated as intrinsically typed with respect to a context and an output type, contexts are ordered lists (this is important for our definition of dynamism context below). For bare preorder type theory, the only types are base types, and the only terms are variables and applications of uninterpreted function symbols (whose rule we omit). In the extended version [20], we give a precise definition of a *signature* specifying valid base types, function symbols, and type and term dynamism axioms. A substitution $\Gamma \vdash \Delta$ is defined as usual as giving, for every typed variable in the output context, a term of that type relative to the input context.

Next, we discuss the new judgments of type dynamism, dynamism contexts, and term

$$\frac{}{A \sqsubseteq A} \qquad \frac{A \sqsubseteq B \qquad B \sqsubseteq C}{A \sqsubseteq C} \qquad \frac{}{\cdot : \cdot \sqsubseteq \cdot} \qquad \frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad A \sqsubseteq A'}{(\Phi, x \sqsubseteq x' : A \sqsubseteq A') : \Gamma, x : A \sqsubseteq \Gamma', x' : A'}$$

Figure 3 Type and Context Dynamism

dynamism. A type dynamism judgment (Figure 3) $A \sqsubseteq B$ relates two well-formed types, and is read as "A is less dynamic than B". In preorder type theory, the only rules are reflexivity and transitivity, making type dynamism a preorder, and axioms from a signature(omitted).

The remaining rules in Figure 3 define *type dynamism contexts* Φ, which are used in the definition of term dynamism. While terms are indexed by a type and a typing context, term dynamism judgments $\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'$ are indexed by two terms $\Gamma \vdash t : A$ and $\Gamma' \vdash t' : A'$, such that $A \sqsubseteq A'$ (A is less dynamic than A') and Γ is less dynamic than Γ'. Thus, we require a judgment $\Phi : \Gamma \sqsubseteq \Gamma'$, which lifts type dynamism to contexts pointwise (for any $x : A \in \Gamma$, the corresponding $x' : A' \in \Gamma'$ satisfies $A \sqsubseteq A'$). This uses the structure of Γ and Γ' as ordered lists: a dynamism context $\Phi : \Gamma \sqsubseteq \Gamma'$ implies that Γ and Γ' have the same length and associates variables based on their order in the context, so that Φ is uniquely determined by Γ and Γ'; this is sufficient because of an admissible exchange rule for terms. We notate dynamism contexts to evoke a logical relations interpretation of term dynamism: under the conditions that $x_1 \sqsubseteq x_1' : A_1 \sqsubseteq A_1', \ldots$ then we have that $t \sqsubseteq t' : B \sqsubseteq B'$.

The term dynamism judgment admits constructions (Figure 4) corresponding to both the structural rules of terms and the preorder structure of type dynamism, beginning from arbitrary term dynamism axioms (see the extended version [20] for a formal definition). First, there is a rule (TmPrec-Var) that relates variables. Next there is a *compositionality* rule (TmPrec-Comp) that allows us to prove dynamism judgments by breaking terms down into components. We elide the definition of *substitution dynamism* $\Phi \vdash \gamma \sqsubseteq \gamma' : \Psi$, which is pointwise term dynamism. Last, we add an appropriate form of reflexivity (TmPrec-Refl) and transitivity (TmPrec-Trans) as rules, whose well-formedness depends on the reflexivity and transitivity of type dynamism. While the reflexivity rule is intuitive, the transitivity rule is more complex. Consider an example where $A \sqsubseteq A' \sqsubseteq A''$ and $B \sqsubseteq B' \sqsubseteq B''$:

$$\frac{x \sqsubseteq x' : A \sqsubseteq A' \vdash t \sqsubseteq t' : B \sqsubseteq B' \qquad x' \sqsubseteq x'' : A' \sqsubseteq A'' \vdash t' \sqsubseteq t'' : B' \sqsubseteq B''}{x \sqsubseteq x'' : A \sqsubseteq A'' \vdash t \sqsubseteq t'' : B \sqsubseteq B''}$$

In a logical relations interpretation of term dynamism, we would have relations $\sqsubseteq_{A,A'}$, $\sqsubseteq_{A',A''}$, $\sqsubseteq_{A,A''}$ and similarly for the B's, and the term dynamism judgment of the conclusion would be interpreted as "for any $u \sqsubseteq_{A,A''} u''$, $t[u/x] \sqsubseteq_{B,B''} t''[u''/x'']''$. However, we could only instantiate the premises of the judgment if we could produce some middle u' with $u \sqsubseteq_{A,A'} u' \sqsubseteq_{A',A''} u''$. In such models, a middle u' must *always* exists, because an implicit condition of the transitivity rule is that $\sqsubseteq_{A,A''}$ is the relation composite of $\sqsubseteq_{A,A'}$ and $\sqsubseteq_{A',A''}$ (the composite exists by type dynamism transitivity, and type dynamism witnesses are unique in PTT (thin in the semantics)). PTT itself does not give a term for this u', but the upcasts and downcasts in gradual type theory do (take it to be $\langle A' \leftharpoonup A \rangle u$ or $\langle A' \leftarrow A'' \rangle u''$).

Sometimes it is convenient to use the same variable name at the same type in both t and t', so we sometimes write $x : A$ in a dynamism context for $x \sqsubseteq x : A \sqsubseteq A$, and write Γ for $x_i \sqsubseteq x_i : A_i \sqsubseteq A_i$ for all $x_i : A_i$ in Γ. Similarly, we write A as the conclusion of a dynamism judgment for $A \sqsubseteq A$, so $\Gamma \vdash t \sqsubseteq t' : A$ means $\Gamma \sqsubseteq \Gamma \vdash t \sqsubseteq t' : A \sqsubseteq A$.

Gradual Type Theory Preorder Type Theory gives us a simple foundation with which to build Gradual Type Theory in a modular way: we can characterize different aspects of

$$\frac{x \sqsubseteq x' : A \sqsubseteq A' \in \Phi}{\Phi \vdash x \sqsubseteq x' : A \sqsubseteq A'}\ \textsc{TmPrec-Var} \qquad \frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad \Psi \vdash \gamma \sqsubseteq \gamma' : \Phi}{\Psi \vdash t[\gamma] \sqsubseteq t'[\gamma'] : A \sqsubseteq A'}\ \textsc{TmPrec-Comp}$$

$$\frac{\Gamma \vdash t : A \qquad \Phi : \Gamma \sqsubseteq \Gamma}{\Phi \vdash t \sqsubseteq t : A \sqsubseteq A}\ \textsc{TmPrec-Refl} \qquad \frac{\begin{array}{c}\Phi : \Gamma \sqsubseteq \Gamma' \vdash t \sqsubseteq t' : A \sqsubseteq A' \\ \Phi' : \Gamma' \sqsubseteq \Gamma'' \vdash t' \sqsubseteq t'' : A' \sqsubseteq A''\end{array}}{\Psi : \Gamma \sqsubseteq \Gamma'' \vdash t \sqsubseteq t'' : A \sqsubseteq A''}\ \textsc{TmPrec-Trans}$$

Figure 4 Primitive Rules of Term Dynamism

$$\frac{\Gamma \vdash t : A \qquad A \sqsubseteq A'}{\Gamma \vdash \langle A' \leftarrowtail A \rangle t : A'}\ \textsc{Upcast} \qquad \frac{\Gamma \vdash t : A' \qquad A \sqsubseteq A'}{\Gamma \vdash \langle A \twoheadleftarrow A' \rangle t : A}\ \textsc{Downcast}$$

$$\text{UR}\ \frac{A \sqsubseteq A'}{x \sqsubseteq x : A \sqsubseteq A \vdash x \sqsubseteq \langle A' \leftarrowtail A \rangle x : A \sqsubseteq A'} \qquad \frac{A \sqsubseteq A'}{x' \sqsubseteq x' : A' \sqsubseteq A' \vdash \langle A \twoheadleftarrow A' \rangle x' \sqsubseteq x' : A \sqsubseteq A'}\ \text{DL}$$

$$\text{UL}\ \frac{A \sqsubseteq A'}{x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \leftarrowtail A \rangle x \sqsubseteq x' : A' \sqsubseteq A'} \qquad \frac{A \sqsubseteq A'}{x \sqsubseteq x' : A \sqsubseteq A' \vdash x \sqsubseteq \langle A \twoheadleftarrow A' \rangle x' : A \sqsubseteq A}\ \text{DR}$$

$$\frac{A \sqsubseteq A'}{x : A \sqsubseteq x : A \vdash \langle A \twoheadleftarrow A' \rangle \langle A' \leftarrowtail A \rangle x \sqsubseteq x : A}\ \textsc{RetractAx} \qquad \frac{}{A \sqsubseteq ?} \qquad \frac{}{\Gamma \vdash \mho_A : A} \qquad \frac{\Phi : \Gamma \sqsubseteq \Gamma}{\Phi \vdash \mho_A \sqsubseteq t : A}$$

Figure 5 Upcasts, Downcasts, Dynamic Type and Type Error

gradual typing, such as a dynamic type, casts, and type errors separately.

We start by defining upcasts and downcasts, using type and term dynamism in Figure 5. Given that $A_0 \sqsubseteq A_1$, the upcast is a function from A_0 to A_1 such that for any $t : A_0$, $\langle A_1 \leftarrowtail A_0 \rangle t$ is the *least dynamic term of type* A_1 *that is more dynamic than* t. The UR rule can be thought of as the "introduction rule", saying $\langle A' \leftarrowtail A \rangle x$ is more dynamic than x, and then UL is the "elimination rule", saying that if some $x' : A'$ is more dynamic than $x : A$, then it is more dynamic than $\langle A' \leftarrowtail A \rangle x$ — since $\langle A' \leftarrowtail A \rangle x$ is the *least* dynamic term with this property. The rules for projections are dual, ensuring that for $x' : A'$, $\langle A \twoheadleftarrow A' \rangle x'$ is the most dynamic term of type A that is less dynamic than x'. In fact, combined with the TmPrec-Trans rule, we can show that it has a slightly more general property: $\langle A' \leftarrowtail A \rangle x$ is not just less dynamic than any term of type A' more dynamic than x, but is less dynamic than any term of type A' *or higher*, i.e. of type $A'' \sqsupseteq A'$.

As we will discuss in Section 3, these rules allow us to prove that the pair of the upcast and downcast form a *Galois connection* (adjunction), meaning $\langle A' \leftarrowtail A \rangle \langle A \twoheadleftarrow A' \rangle t \sqsubseteq t$ and $t \sqsubseteq \langle A \twoheadleftarrow A' \rangle \langle A' \leftarrowtail A \rangle t$. However in programming practice, the casts satisfy the stronger condition of being a *Galois insertion*, in which the left adjoint, the downcast, is a *retract* of the upcast, meaning $t \sqsupseteq\sqsubseteq \langle A \twoheadleftarrow A' \rangle \langle A' \leftarrowtail A \rangle t$. We can restrict to Galois insertions by adding the *retract axiom* RetractAx. Most theorems of gradual type theory do not require it, though this axiom is satisfied in all models of preorder type theory in Section 6.

The remaining rules in Figure 5 define the dynamic type and type errors, which are also given a universal property in terms of type and term dynamism. The dynamic type is defined as the most dynamic type. The type error, written as $\mho$, is defined by the fact that it is a constant at every type that is a least element of that type. By transitivity, this further implies that $\mho_A \sqsubseteq t : A \sqsubseteq A'$ for any $A' \sqsupseteq A$.

$$\frac{A \sqsubseteq A' \qquad B \sqsubseteq B'}{A \to B \sqsubseteq A' \to B'} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}$$

$$\frac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \vdash t \sqsubseteq t' : B \sqsubseteq B'}{\Phi \vdash \lambda x : A.t \sqsubseteq \lambda x' : A'.t' : A \to B \sqsubseteq A' \to B'} \qquad \frac{\Phi \vdash t \sqsubseteq t' : A \to B \sqsubseteq A' \to B' \quad \Phi \vdash u \sqsubseteq u' : A \sqsubseteq A'}{\Phi \vdash t\, u \sqsubseteq t'\, u' : A \to B \sqsubseteq A' \to B'}$$

$$\frac{}{\Gamma \vdash (\lambda x : A.t) u \sqsupseteq\sqsubseteq t[u/x] : B} \qquad \frac{}{\Gamma \vdash t \sqsupseteq\sqsubseteq (\lambda x : A.t\, x) : A \to B \sqsubseteq A \to B}$$

Figure 6 Function Type

Next we illustrate how simple *negative* types can be defined in preorder type theory. Figure 6 presents the rules for function types, while the product and unit types are analogous (see the extended version [20]). The type and term constructors are the same as those in the simply typed λ-calculus. For type dynamism, we make every connective *monotone* in every argument, including the function type. For term dynamism, we add two classes of rules. First, there are congruence rules that "extrude" the term constructor rules for the type, which are like a "congruence of contextual approximation" condition. Next, the computational rules reflect the ordinary β, η equivalences as equi-dynamism: we write $\sqsupseteq\sqsubseteq$ to mean a rule exists in each direction (which requires that the types and contexts are also equi-dynamic).

We call the accumulation of all of these connectives *gradual type theory*. In the extended version [20], we define a GTT signature, which gives axioms for base types, function symbols, type dynamism, and term dynamism, which all may make use of the dynamic type, casts, type error, functions, and products types, in addition to the rules of PTT.

3 Theorems and Constructions in Gradual Type Theory

In this section, we discuss the many consequences of the simple axioms of gradual type theory. We show that almost every reduction in an operational presentation of call-by-name gradual typing, and many principles used in optimization of implementations, are justified by the universal property for casts in all types, the β, η rules, and the congruence rules for connectives and terms. Thus, the combination of graduality and η principles is a strong specification for gradual typing and considerably narrows the design space. We summarize these derivations in the following theorem:

▶ **Theorem 1.** *In Gradual Type Theory, all of the following are derivable whenever the upcasts, downcasts are well-formed.*

1. *Universal Property: Casts are unique up to* $\sqsupseteq\sqsubseteq$.
2. *Identity:* $\langle A \leftarrow\!\!\prec A \rangle t \sqsupseteq\sqsubseteq t$ *and* $\langle A \twoheadleftarrow A \rangle t \sqsupseteq\sqsubseteq t$.
3. *Composition:* $\langle A'' \leftarrow\!\!\prec A \rangle t \sqsupseteq\sqsubseteq \langle A'' \leftarrow\!\!\prec A' \rangle \langle A' \leftarrow\!\!\prec A \rangle t$ *and* $\langle A \twoheadleftarrow A'' \rangle t \sqsupseteq\sqsubseteq \langle A \twoheadleftarrow A' \rangle \langle A' \twoheadleftarrow A'' \rangle t$.
4. *Function Cast Reduction:* $\langle A' \to B' \leftarrow\!\!\prec A \to B \rangle t \sqsupseteq\sqsubseteq \lambda x : A'.\langle B' \leftarrow\!\!\prec B \rangle (t(\langle A \twoheadleftarrow A' \rangle x))$ *and* $\langle A \to B \twoheadleftarrow A' \to B' \rangle t \sqsupseteq\sqsubseteq \lambda x : A'.\langle B \twoheadleftarrow B' \rangle (t(\langle A' \leftarrow\!\!\prec A \rangle x))$.
5. *Product Cast Reduction:* $\langle A'_0 \times A'_1 \leftarrow\!\!\prec A_0 \times A_1 \rangle t \sqsupseteq\sqsubseteq (\langle A'_0 \leftarrow\!\!\prec A_0 \rangle \pi_0 t, \langle A'_1 \leftarrow\!\!\prec A_1 \rangle \pi_1 t)$ *and* $\langle A_0 \times A_1 \twoheadleftarrow A'_0 \times A'_1 \rangle t \sqsupseteq\sqsubseteq (\langle A_0 \twoheadleftarrow A'_0 \rangle \pi_0 t, \langle A_1 \twoheadleftarrow A'_1 \rangle \pi_1 t)$.
6. *Adjunction:* $t \sqsubseteq \langle A \twoheadleftarrow A' \rangle \langle A' \leftarrow\!\!\prec A \rangle t$ *and* $\langle A' \leftarrow\!\!\prec A \rangle \langle A \twoheadleftarrow A' \rangle t \sqsubseteq t$, *for which the retract axiom is the converse.*
7. *Cast Congruence:* $x \sqsubseteq y : A \sqsubseteq B \vdash \langle A' \leftarrow\!\!\prec A \rangle x \sqsubseteq \langle B' \leftarrow\!\!\prec B \rangle y : A' \sqsubseteq B'$ *and* $x' \sqsubseteq y' : A' \sqsubseteq B' \vdash \langle A \twoheadleftarrow A' \rangle x' \sqsubseteq \langle B \twoheadleftarrow B' \rangle y' : A \sqsubseteq B$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f \sqsubseteq \lambda x.fx}\quad \dfrac{\dfrac{}{f, x \sqsubseteq x' \vdash f \sqsubseteq f : A \to B \sqsubseteq A \to B}\quad \dfrac{}{f, x \sqsubseteq x' \vdash x \sqsubseteq x' : A \sqsubseteq A'}}{f, x \sqsubseteq x' \vdash x \sqsubseteq \langle A \twoheadleftarrow A' \rangle x' : A \sqsubseteq A}}{f, x \sqsubseteq x' \vdash fx \sqsubseteq f(\langle A \twoheadleftarrow A' \rangle x') : B \sqsubseteq B}}{f, x \sqsubseteq x' \vdash fx \sqsubseteq \langle B' \leftarrowtail B \rangle (f(\langle A \twoheadleftarrow A' \rangle x')) : B \sqsubseteq B'}}{\lambda x.fx \sqsubseteq \lambda x' : A'.\langle B' \leftarrowtail B \rangle (f(\langle A \twoheadleftarrow A' \rangle x'))}}{f \vdash f \sqsubseteq \lambda x' : A'.\langle B' \leftarrowtail B \rangle (f(\langle A \twoheadleftarrow A' \rangle x')) : A \to B \sqsubseteq A' \to B'}}{f : A \to B \vdash \langle A' \to B' \leftarrowtail A \to B \rangle f \sqsubseteq \lambda x' : A'.\langle B' \leftarrowtail B \rangle (f(\langle A \twoheadleftarrow A' \rangle x')) : A' \to B'}$$

Figure 7 Function Upcast Implementation (one case)

8. *Errors:* $\langle A' \leftarrowtail A \rangle \mho_A \sqsupseteq\sqsubseteq \mho_{A'}$*, and by the retract axiom* $\langle A \twoheadleftarrow A' \rangle \mho_A \sqsupseteq\sqsubseteq . \mho_{A'}$*.*
9. *Equi-dynamism implies isomorphism: If* $A \sqsupseteq\sqsubseteq B$*, then* A *is isomorphic to* B*.*

We present one example proof (most of the rest can be found in the extended version), and give some intuition for the others based on the defining properties of upcasts and downcasts as meets and joins. Part 1 says that this specification defines them *uniquely*, which we can prove by duplicating the rules for upcasts/downcasts and showing the two are $\sqsupseteq\sqsubseteq$. First, identity (2) and composition (3) are intuitive consequences that are sometimes operational reductions. Part 2 says the cast from a type to itself is the identity function and is easily justified by the specification: given $t : A$, t itself is the least dynamic element of A that is at least as dynamic as t. Part 3 says that if $A \sqsubseteq A' \sqsubseteq A''$ then casts between A, A'' factor through A'. This is important operationally, justifying the common rule $\langle A \to B \twoheadleftarrow ? \rangle t \mapsto \langle A \to B \twoheadleftarrow ? \to ? \rangle \langle ? \to ? \twoheadleftarrow ? \rangle t$ which says that casting to a function type first does the first order check to make sure t is a function, and then performs the checking of the function's behavior. More generally, it implies that casts from A to B commute over the dynamic type, e.g. $\langle ? \leftarrowtail B \rangle \langle B \leftarrowtail A \rangle x \sqsupseteq\sqsubseteq \langle ? \leftarrowtail A \rangle x$—intuitively, if casts only perform checks, and do not change values, then a value's representation in the dynamic type should not depend on how it got there.

Next, consider the function contract reduction (4) (5 is similar). These equivalences are the standard "wrapping" implementations from [9, 8]: a function upcast uses the downcast on inputs and upcast on outputs and vice-versa for the downcast. This shows that the standard implementation is in fact the *unique* implementation to satisfy soundness and graduality. We present one of the cases for upcasts in Figure 7; the other 3 proofs are similar. First, we use the UL rule to reduce to showing the wrapping implementation is more dynamic than f itself. Then we use transitivity to η-expand f and then apply the λ congruence rule. Then we know the upcast $\langle B' \leftarrowtail B \rangle \cdot$ makes the term more dynamic and then apply function application congruence and use DR to show that the downcast of x' is still more dynamic than x. The other cases and the product cases follow by similar proofs.

Next, as mentioned previously, the adjunction property (6) shows that the upcast and downcast are a Galois connection with the upcast as the upper/left adjoint. This tells us that given $A \sqsubseteq A'$, the "round-trip" from A' down to A and back results in a less dynamic term and the other round-trip results in a more dynamic term. In programming practice, we expect the round trip from A to A' and back to be in fact an identity, as in the above retract axiom RetractAx. This theorem is the basis for our model (Section 6) where we *define* type dynamism as a pair of functions with these properties.

Next, it is important for proving the gradual guarantee [23] that all term constructors are congruences with respect to type and term dynamism. While for types like functions and

products these are primitive rules, for upcasts and downcasts congruence is derivable (7).

Error strictness (8) states that upcasts and downcasts are *strict* with respect to the type error $\mho$. The upcast preserves $\mho$ because it is a left/upper adjoint and therefore preserves colimits/joins like $\mho$. The proof that the downcast preserves $\mho$ is less modular, and uses the upcast, the retract axiom, and strictness of the upcast.

Finally, because types A and B in gradual type theory can be related both by type dynamism $A \sqsubseteq B$ and by functions $A \to B$, there are two reasonable notions of equivalence of types[1]. First, *equi-dynamism* $A \sqsupseteq\sqsubseteq B$ means $A \sqsubseteq B$ and $B \sqsubseteq A$. Second, isomorphism means functions $f : A \to B$ and $g : B \to A$ such that $f \circ g \sqsupseteq\sqsubseteq (\lambda x : B.x)$ and $g \circ f \sqsupseteq\sqsubseteq (\lambda x : A.x)$. Part 9 gives an isomorphism for any equi-dynamic types.

The converse, isomorphic types are equi-dynamic, does not hold by design, because it does not match gradual typing practice. Gradually typed languages typically have *disjointness* of connectives as operational reductions; for example, disjointness of products and functions can be expressed by an axiom $\langle (C \times D) \twoheadleftarrow ? \rangle \langle ? \leftarrowtail (A \to B) \rangle x \sqsubseteq \mho$ which says that casting a function to a product errors. This axiom is incompatible with isomorphic types being equi-dynamic, because a function type *can* be isomorphic to a product type (e.g. $X \to Y \cong (X \to Y) \times 1$), and for equi-dynamic types A and B, a cast $\langle B \twoheadleftarrow ? \rangle \langle ? \leftarrowtail A \rangle x$ should succeed, not fail (if it fails, then every term of A, B equal $\mho$; see the extended version). That is, disjointness axioms make equi-dynamism an intensional property of the representation of a type, and therefore stronger than isomorphism. Nonetheless, the basic rules of gradual type theory do not imply disjointness; in Section 6, we discuss a countermodel.

4 Categorical Semantics

Next, we define what a category-theoretic model of preorder and gradual type theory is, and prove that PTT/GTT are *internal languages* of these classes of models by proving soundness and completeness (i.e. initiality) theorems. This alternative axiomatic description of PTT/GTT is a useful bridge between the syntax and the concrete models presented in Section 6. The models are in *preorder categories*, which are categories internal to the category of preorders.[2] A preorder category is a category where the set of all objects and set of all arrows are each equipped with a preorder (a reflexive, transitive, but not necessarily anti-symmetric, relation). Furthermore the source, target, identity and composition functions are all *monotone* with respect to these orderings. A preorder category is also a double category where one direction of morphism is thin. Intuitively, the preorder of objects represents types and type dynamism, while the preorder of morphisms represents terms and term dynamism, and we reuse the notation $\sqsubseteq$ for the orderings on objects and morphisms.

While the axioms of a preorder category are *similar to* the judgmental structure of preorder type theory, in a preorder category, morphisms have *one* source object and one target object, whereas in preorder type theory, terms have an entire *context* of inputs and one output. This is a standard mismatch between categories and type theories, and is classically resolved by assuming that models have product types and using categorical products to interpret the context [14]. However, we take a more modern *multicategorical* view, in which our notion of model will axiomatize algebraically a notion of morphism with many inputs.

[1] Corresponding to the two notions of isomorphism in double categories

[2] To avoid confusion, these are not categories that happen to be preorders (thin categories) and these are not categories *enriched* in the category of preorders, where the hom-sets between two objects are preordered, but the objects are not.

Using terminology from [4], we define a model of preorder type theory as a "virtually" cartesian preorder category, which does not necessarily have product *objects*, but whose morphisms' source is a "virtual" product of objects, i.e. a context. In the extended version [20], we prove soundness and completeness of PTT for VCP categories.

▶ **Definition 2** (Virtually Cartesian Preorder Category)**.** A virtually cartesian preorder category (VCP category) $\mathbb{C}$ consists of a preordered set of "objects" $\mathbb{C}_0$, a preordered set of "multiarrows" $\mathbb{C}_1$, monotone functions "source" $s : \mathbb{C}_1 \to \mathrm{Ctx}(\mathbb{C})_0$, "target" $t : \mathbb{C}_1 \to \mathbb{C}_0$, "projection" $x : \mathrm{context}(\mathbb{C})_0 \times \mathbb{C}_0 \times \mathrm{Ctx}(\mathbb{C}) \to \mathbb{C}_1$ and composition $\circ : \mathbb{C}_1 \times_{\mathrm{Ctx}(\mathbb{C})_0} \mathrm{Ctx}(\mathbb{C})_1 \to \mathbb{C}_1$. Here $\mathrm{Ctx}(\mathbb{C})_0$ is the set of lists of objects preordered pointwise, and a substitution $\gamma \in \mathrm{Ctx}(\mathbb{C})_1(\Gamma; B_0, \ldots, B_n)$ consists of a multiarrow $\gamma(i) \in \mathbb{C}_1(\Gamma; B_i)$ for each $i \in 0, \ldots, n$, also preordered pointwise, and with composition defined in the same way as syntactic substitutions. Additionally these satisfy associativity and unitality laws (see the extended version [20]).

Gradual Typing Structures Next, we describe the additional structure on a VCP category to model full gradual type theory: casts are modeled by an *equipment* [22], a dynamic type by a greatest object, and the type error by a least element of every hom-set.

▶ **Definition 3** (Gradual Structure on a VCP category)**.**

1. A VCP category $\mathbb{C}$ is an *equipment* if for every $A \sqsubseteq B$, there exist morphisms $u_{A,B} \in \mathbb{C}(A, B)$ and $d_{A,B} \in \mathbb{C}(B, A)$ such that $u_{A,B} \sqsubseteq \mathrm{id}_B$ and $\mathrm{id}_A \sqsubseteq u_{A,B}$ and $d_{A,B} \sqsubseteq \mathrm{id}_B$ and $\mathrm{id}_A \sqsubseteq d_{A,B}$. An equipment is *coreflective* if also $d_{A,B} \circ u_{A,B} \sqsubseteq \mathrm{id}_A$.
2. A greatest object in a VCP category $\mathbb{C}$ is a greatest element of the preorder of objects $\mathbb{C}_0$.
3. A VCP category $\mathbb{C}$ has local bottoms if every hom set $\mathbb{C}(A_1, \ldots, A_n; B)$ has a least element $\bot$ and for every substitution γ we have $\bot \circ \gamma \sqsupseteq\sqsubseteq \bot$.

Next, we define a *cartesian closed VCP category*, which will model negative function and product types. We present the definition for function types in detail; a definition of a cartesian VCP category is in the extended version [20]. A *cartesian closed VCP category* is a VCP category with a choice of both cartesian and closed structure.

▶ **Definition 4** (Closed VCP Category)**.** A Closed VCP category is a VCP category $\mathbb{C}$ with a monotone function on objects $\to : \mathbb{C}_0^2 \to \mathbb{C}_0$ making for every pair of objects $X, Y \in \mathbb{C}$ an "exponential" object $X \to Y$ with a monotone function $\lambda : \mathbb{C}(\Gamma, X; Y) \to \mathbb{C}(\Gamma; X \to Y)$ that is *natural* in an appropriate sense, with a morphism $\mathrm{app} \in \mathbb{C}(X \to Y, X; Y)$ such that the function given by $f \mapsto \mathrm{app} \circ (f, x(X))$ is an inverse to λ up to $\sqsupseteq\sqsubseteq$.

In the extended version [20], we prove the following, where a *GTT category* is a cartesian closed VCP coreflective equipment with a greatest object and local bottoms.

▶ **Definition 5** (Interpretation of Gradual Type Theory/Soundness)**.** For any GTT signature Σ and GTT category $\mathbb{C}$ and interpretation $(\!|\cdot|\!) : \Sigma \to \mathbb{C}$ of the base types and function symbols in Σ such that all type and term dynamism axioms in Σ are true in $\mathbb{C}$, there is an extension of $(\!|\cdot|\!)$ to an interpretation of all types and terms of GTT generated by Σ, such that all derivable type and term dynamism theorems are true in $\mathbb{C}$.

▶ **Theorem 6** (Completeness of GTT Category Semantics)**.** *For any GTT signature Σ, for any GTT_Σ types A, B if for every interpretation $(\!|\cdot|\!) : \Sigma \to \mathbb{C}$, $[\![A]\!] \sqsubseteq [\![B]\!]$ holds, then $A \sqsubseteq B$ is derivable. For any GTT_Σ contexts $\Phi : \Gamma \sqsubseteq \Gamma'$, types $A \sqsubseteq A'$, and terms $\Gamma \vdash t : A$ and $\Gamma' \vdash t' : A'$, if for every interpretation $[\![t]\!] \sqsubseteq [\![t']\!]$, then $\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'$ derivable.*

As usual, the proof of completeness is by building a GTT category from the syntax such that the true dynamism theorems are precisely the derivable ones.

5 Semantic Contract Interpretation

As a next step towards constructing specific GTT categories, we define a general *contract construction* that provides a semantic account of the "contract interpretation" of gradual typing, which models a gradual type by a pair of casts. The input to our contract construction is a locally thin 2-category $\mathbb{C}$, whose objects and arrows should be thought of as the types and terms of a programming language, and each hom-set $\mathbb{C}(A, B)$ is ordered by an "approximation ordering", which is used to define term dynamism in our eventual model. We require each hom-set to have a least element (the type error), and the category to be cartesian closed (function and product types/contexts). The contract construction then "implements" gradual typing using the morphisms of the non-gradual "programming language" $\mathbb{C}$.

Coreflections To build a GTT model from $\mathbb{C}$, we need to choose an interpretation of type dynamism (the ordering on objects of the VCP category) that induces appropriate casts, which we know by Theorem 1.6 must be Galois connections that satisfy the retract axiom. Such Galois connections are called Galois insertions (in order theory), coreflections (in category theory) and embedding-projection pairs (in domain theory). Since type dynamism judgments must induce a coreflection, we will construct a model where the semantics of a type dynamism judgment $A \sqsubseteq B$ is literally a coreflection. However, there can be many different coreflections between two objects of our 2-category $\mathbb{C}$, so this first step of our construction does not produce a preorder category, where type dynamism is an *ordering*, but rather a *double category*. Double categories generalize preorder categories in the same way that categories generalize preorders: the ordering on objects is generalized to proof-relevant data specifying a second class of *vertical morphisms*, and the ordering on terms becomes a notion of 2-dimensional "square" between morphisms. In the model we build from $\mathbb{C}$, the vertical morphisms will model type dynamism and be coreflections, while the *(horizontal)* morphisms of a preorder category will be arbitrary morphisms of $\mathbb{C}$ and model terms. We still require only double categories that are *locally thin*, in that there is at most one 2-cell filling in any square. Thus, the first step of our contract construction can be summarized as creating a double category that is an equipment with the retract property, i.e. a double category modeling upcasts and downcasts, a slight variation on a theorem in [22]:

▶ **Definition 7** (Equipment of Coreflections [22])**.** Given a 2-category $\mathbb{C}$ we construct a (double category) equipment CoReflect($\mathbb{C}$) as follows. Its object category has $\mathbb{C}_0$ as objects and coreflections in $\mathbb{C}$ as morphisms. Horizontal morphisms are given by morphisms in $\mathbb{C}$ and a 2-cell from $f : A \to B$ to $f' : A' \to B'$ along $(u_A, d_A) : A \triangleleft A'$ and $(u_B, d_B) : B \triangleleft B'$ is a 2-cell in $\mathbb{C}$ from $u_B \circ f$ to $f' \circ u_A$ or equivalently from $f \circ d_A$ to $d_B \circ f'$. From a vertical arrow (u, d), the upcast is u and the downcast is d.

As is well-known in domain theory, any mixed-variance functor preserves coreflections [31, 26], so the product and exponential functors of $\mathbb{C}$ extend to be functorial also in vertical arrows. This produces the classic "wrapping" construction familiar from higher-order contracts [9]:$(u, d) \to (u', d') = (d \to u', u \to d')$

Vertical Slice Category The double category CoReflect($\mathbb{C}$) is not yet a model of gradual type theory for two reasons. First, gradual type theory requires a dynamic type: every type should have a canonical coreflection into a specific type. Second, type dynamism in GTT is *proof-irrelevant*, because the rules do not track different witnesses of $A \sqsubseteq B$, but there may be different coreflections from A to B. It turns out that we can solve both problems at once by taking what we call the "vertical slice" category over an object $D \in$ CoReflect($\mathbb{C}$)

that is rich enough to serve as a model of the dynamic type. In $\mathrm{CoReflect}(\mathbb{C})/D$, the objects are not just an object A of $\mathbb{C}$, but an object *with* a vertical morphism into D, in this case a coreflection written $(u_A, d_A) : A \triangleleft D$.[3] Thus, gradual types are modeled as coreflections into the dynamic type, analogous to Scott's "retracts of a universal domain" [21]. Then a vertical arrow from $(u_A, d_A) : A \triangleleft D$ to $(u_B, d_B) : B \triangleleft D$ is a coreflection $(u_{A,B}, d_{A,B}) : A \triangleleft B$ that *factorizes* $u_A = u_B \circ u_{A,B}$ and $d_A = d_{A,B} \circ d_B$: this means the enforcement of A's type can be thought of as also enforcing B's type. Since upcasts are monomorphisms and downcasts are epimorphisms, this factorization is *unique* if it exists, so there is at most one vertical arrow between any two objects of $\mathrm{CoReflect}(\mathbb{C})/D$. Further, the identity coreflection $(\mathrm{id}, \mathrm{id}) : D \triangleleft D$ is a vertically greatest element since any morphism is factorized by the identity.

▶ **Definition 8** (Vertical Slice Category)**.** Given any double category $\mathbb{E}$ and an object $D \in \mathbb{E}$, we can construct a double category $\mathbb{E}/D$ by defining $(\mathbb{E}/D)_0$ to be the slice category $\mathbb{E}_0/D$, a horizontal morphism from $(c : A \triangleleft D)$ to $d : B \triangleleft D$ to be a horizontal morphism from A to B in $\mathbb{E}$, and the 2-cells are similarly inherited from $\mathbb{E}$.

Next consider cartesian closed structure on $\mathrm{CoReflect}(\mathbb{C})/D$. The action of $\to$ (respectively $\times, 1$) on objects is given by composition of the action in $\mathrm{CoReflect}(\mathbb{C})$ $(u, d) \to (u', d')$ with an *arbitrary choice* of "encoding" of the "most dynamic function type" $(u_\to, d_\to) : (D \to D) \triangleleft D$. In most of the models we consider later, D is a sum and this coreflection simply projects out of the corresponding case, failing otherwise. This reflects the separation of the function contract into "higher-order" checking $(u, d) \to (u', d')$ and "first-order tag" checking $(u_\to, d_\to)$ that has been observed in implementations [11].

Finally, we construct a *virtually* cartesian model from a cartesian model by defining $\mathrm{Virt}(\mathbb{C})_1(A_1, \ldots, A_n; B) = \mathbb{C}_1(A_1 \times \cdots \times A_n; B)$. Combining these constructions, we produce:

▶ **Theorem 9** (Contract Model of Gradual Typing)**.** *If $\mathbb{C}$ is a locally thin cartesian closed 2-category with local $\perp$s, then for any object $D \in \mathbb{C}$ with chosen coreflections $c_\to : (D \to D) \triangleleft D$, $c_\times : (D \times D) \triangleleft D$, and $c_1 : 1 \triangleleft D$, then $(\mathit{Virt}(\mathit{CoReflect}(\mathbb{C})/\mathit{id}_d), c_\to, c_\times, c_1)$ is a GTT category.*

6 Concrete Models

Next, we produce some concrete models by instantiating Theorem 9.

There are two models based on domains that are operationally *inadequate* in that they identify the dynamic type error and diverging programs: term dynamism is given by the "definedness ordering" of domain theory. Both are based on the 2-category of domains (ω-cppos), continuous functions, and the domain ordering on functions, with different choices of universal domain. The first is merely a new presentation of Dana Scott's classical models of untyped λ-calculus, showing that Scott's model is already a model of gradual typing [21]. That is, we find the solution to the recursive domain equation $D \cong \mathbb{N}_\perp \oplus (D \times D) \oplus (D \to D)$ where $\oplus$ is the coalesced sum of domains. The classical technique for solving this equation naturally produce the required coreflections $(D \times D) \triangleleft D$ and $(D \to D) \triangleleft D$. The second is a variation where product and function types have overlapping representation, showing that the product and function types cannot be proven disjoint in gradual type theory. To do this, we construct a universal domain as a *product* of basic connectives rather than a sum: $D' \cong \mathbb{N}_\perp \times (D' \times D') \times (D' \to D')$. This is a kind of "coinductive"/"object-oriented" dynamic type: an element of the dynamic type responds to messages (given by the projections), and

[3] We do not write $A \sqsubseteq D$ because coreflections are not a preorder.

if it "doesn't implement" a message it returns $\bot$. Then $\langle (? \times ?) \twoheadleftarrow ? \rangle \langle ? \leftharpoondown (? \to ?) \rangle x \not\equiv \mho$ the domain has elements that are non-trivial in both the $D \times D$ and $D \to D$ positions.

Next, to produce a domain theoretic model that is adequate, we want a notion of domain that has, in addition to the definedness ordering needed for solving recursive domain equations, a *separate* notion of type error and error-approximation ordering. This can be accomplished using "pointed domain preorders", which are both domains and pointed preorders such that the preordering is an admissible relation.

▶ **Definition 10** (Pointed Domain Preorder). A pointed domain preorder is a set X with two orderings $\leq$ and $\sqsubseteq$ such that $(X, \leq)$ is an ω-cpo with $\leq$-least element $\bot$ and $\sqsubseteq$ is a preorder closed under limits of $\leq$-ω-chains with least element $\mho$. A continuous function of is a function of the underlying sets that is continuous with respect to $\leq$ and monotone with respect to $\sqsubseteq$.

This category extends to a locally thin 2-category in two different ways: we use the domain order $\leq$ to solve recursive domain equations and the error ordering $\sqsubseteq$ to model term dynamism. The 2-category with the domain ordering satisfies all the criteria of [26] and so we can construct a model by solving essentially the same equation as in Scott's model: $D \cong \mathbb{N}_{\bot,\mho} \oplus (D \times D) \oplus (D \to D)$. Then we can construct coreflections (with respect to $\sqsubseteq$) $(D \times D) \triangleleft D$ and $(D \to D) \triangleleft D$: the downcast produces $\mho$ unless it is the $D \times D$ (respectively $D \to D$) case and the $1 \triangleleft D$ is the unique coreflection between those objects.

7 Related and Future Work

Our logic and semantics of type and term dynamism builds on the formulation introduced with the gradual guarantee in [23], but the rules of our system differ in several ways. First, we only allow casts that are either upcasts or downcasts (as defined by type dynamism), whereas their system allows for a more liberal "compatibility" condition. Accordingly our rules of dynamism for casts are slightly different, but where it makes sense, the rules of the two systems are interderivable. Second, our system also includes the β, η equivalences as equi-dynamism axioms, making term dynamism more semantic.

Our semantic model of contracts as coreflections has precedent in much previous work, though we are the first to identify the relationship to gradual typing's notions of type and term dynamism in addition to the connections to domain theory described in Section 6 .

Henglein's work [11] on dynamic typing defines casts that are retracts to the dynamic type, introduced the upcast-followed-by-downcast factorization that we use here, and defines a syntactic rewriting relation similar to our term dynamism rules. Further they define a "subtyping" relation that is the same as type dynamism and characterize it by a semantic property analogous to the semantics of type dynamism in our contract model. The upcast-downcast factorization of an arbitrary cast is superficially similar to the work on triple casts in [25], which collapse a sequence of casts starting at A and ending at B into a downcast to $A \sqcap B$ followed by an upcast to B. But note that this factorization is opposite (downcast and then upcast), and the upcast-downcast factorization requires only a dynamic type, while the converse requires an appropriate middle type, similarly to *image factorization*. Moreover, [10] shows that the correctness of factorization through $A \sqcap B$ is not always possible.

Findler and Blume's work on contracts as *pairs of projections* [8] is also similar. There a contract is defined in an untyped language to be given by a *pair* of functions that divide enforcement of a type between the a "positive" component that checks the term and a "negative" component that checks the continuation, naturally supporting a definition of *blame* when a contract is violated. We give no formal treatment of blame in this paper, but our separation into upcasts and downcasts naturally supports a definition of blame analogous to

theirs. In their paper, each component c is idempotent and satisfies $c \sqsubseteq \mathrm{id}$. Their work is fundamentally untyped so a direct comparison is difficult.

Recent work on interoperability in a (non-gradual) dependently typed language [5, 6] defines casts as "partial type equivalences" between types, which are defined as a pair of terms $f : A \to B$, $g : B \to A$ satisfying projection in *both* directions: $f \circ g \sqsubseteq \mathrm{id}$ and $g \circ f \sqsubseteq \mathrm{id}$. This does not model type dynamism, but rather the notion of "general" cast that is not necessarily an upcast or a downcast. Using our decomposition of general casts into an upcast followed by a downcast, we can prove in our logic that any general cast is a partial type equivalence. Their work also identifies Galois connections/insertions as being a possible model of upcasts and downcasts, but they do not develop the idea further.

There are two recent proposals for a more general theory of gradual typing: Abstracting Gradual Typing (AGT) [10] and the Gradualizer [3]. Broadly, their systems and ours are similar in that type dynamism and graduality are central and a gradually typed language is constructed from a statically typed language. Gradual type theory is quite different in that it is based on an axiomatic semantics, whereas both of theirs are based on operational semantics. As such our notion of gradual type soundness is stronger than theirs: we assert program equivalences whereas their soundness theorem is related to the syntactic type soundness theorem of the static language. Their systems also develop a *surface syntax* for gradually typed languages (including implicit casts and gradual type checking), whereas our logic here only applies to the *runtime semantics* of the language. Finally, AGT is based on abstract interpretation and uses a Galois insertion between gradual types and sets of static types, but we do not see a precise relationship to our use of coreflections.

Relative to this related work, we believe the axiomatic specification of casts via a universal property relative to dynamism is a new idea in gradual typing, as is our categorical semantics and the presentation of the contract interpretation as a model construction.

In this paper we have shown that the combination of soundness and graduality produces strong specifications for call-by-name gradual typing implementations. However so far we have only validated this by denotational semantics, and we plan to develop *operational models* of this kind of gradual type theory where term dynamism is modeled by a type of contextual approximation. We also will investigate extensions to richer languages. First, we would like to develop a similar theory for call-by-value gradual typing, as every gradually typed language in use today is call-by-value. We plan to build on existing work on categorical semantics and universal properties of types in call-by-value [16, 27]. The combination of gradual typing and parametric polymorphism has proven quite complex [17, 19, 1, 13]. If we could show that the combination of graduality with parametricity has a unique implementation, as we have shown here for simple typing, it would provide a strong semantic justification for a design.

References

1 A. Ahmed, D. Jamner, J. G. Siek, and P. Wadler. Theorems for free for free: Parametricity, with and without types. In *International Conference on Functional Programming (ICFP)*, 2017.

2 F. Bañados Schwerter, R. Garcia, and E. Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 283–295, 2014.

3 M. Cimini and J. G. Siek. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 789–803, 2017.

4 G. S. H. Cruttwell and M. A. Shulman. A unified framework for generalized multicategories. *Theory and Applications of Categories*, 24(21), 2009.

5 P.-E. Dagand, N. Tabareau, and E. Tanter. Partial type equivalences for verified dependent interoperability. In *International Conference on Functional Programming (ICFP)*, 2016.

6 P.-E. Dagand, N. Tabareau, and É. Tanter. Foundations of Dependent Interoperability. working paper or preprint, 2017.

7 M. Felleisen. On the expressive power of programming languages. *ESOP'90*, 1990.

8 R. Findler and M. Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, Apr. 2006.

9 R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, Sept. 2002.

10 R. Garcia, A. M. Clark, and E. Tanter. Abstracting gradual typing. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

11 F. Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Programming*, 22(3):197–230, 1994.

12 A. Igarashi, P. Thiemann, V. Vasconcelos, and P. Wadler. Gradual session types. In *International Conference on Functional Programming (ICFP)*, 2017.

13 Y. Igarashi, T. Sekiyama, and A. Igarashi. On polymorphic gradual typing. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom*, 2017.

14 J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

15 N. Lehmann and E. Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017.

16 P. B. Levy. *Call-by-Push-Value*. Ph. D. dissertation, Queen Mary, University of London, London, UK, Mar. 2001.

17 J. Matthews and A. Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*, Mar. 2008.

18 E. Moggi. Notions of computation and monads. *Inform. And Computation*, 93(1), 1991.

19 G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. In *International Conference on Functional Programming (ICFP)*, pages 135–148, Sept. 2009.

20 M. S. New and D. L. Licata. Call-by-name gradual type theory (extended version). http://maxsnew.github.io/docs/cbn-gtt-ext.pdf.

21 D. Scott. Data types as lattices. *Siam Journal on computing*, 5(3):522–587, 1976.

22 M. Shulman. Framed bicategories and monoidal fibrations. *Theory and Applications of Categories*, 20(18):650–738, 2008.

23 J. Siek, M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages*, SNAPL 2015, 2015.

24 J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (Scheme)*, pages 81–92, Sept. 2006.

25 J. G. Siek and P. Wadler. Threesomes, with and without blame. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 365–376, 2010.

26 M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4), 1982.

27 S. Staton and P. B. Levy. Universal properties of impure programming languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2013.

28 S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium (DLS)*, pages 964–974, Oct. 2006.

29 S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 2008.

30 P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, Mar. 2009.

31 M. Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 8(1):13 – 30, 1979.

32 R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, 2011.