Gradual Type Theory

MAX S. NEW, Northeastern University, USA
DANIEL R. LICATA, Wesleyan University, USA
AMAL AHMED, Northeastern University, USA and Inria Paris, France

Gradually typed languages are designed to support both dynamically typed and statically typed programming styles while preserving the benefits of each. While existing gradual type soundness theorems for these languages aim to show that type-based reasoning is preserved when moving from the fully static setting to a gradual one, these theorems do not imply that correctness of type-based refactorings and optimizations is preserved. Establishing correctness of program transformations is technically difficult, because it requires reasoning about program equivalence, and is often neglected in the metatheory of gradual languages.

In this paper, we propose an *axiomatic* account of program equivalence in a gradual cast calculus, which we formalize in a logic we call *gradual type theory* (GTT). Based on Levy's call-by-push-value, GTT gives an axiomatic account of both call-by-value and call-by-name gradual languages. Based on our axiomatic account we prove many theorems that justify optimizations and refactorings in gradually typed languages. For example, *uniqueness principles* for gradual type connectives show that if the $\beta\eta$ laws hold for a connective, then casts between that connective must be equivalent to the so-called "lazy" cast semantics. Contrapositively, this shows that "eager" cast semantics violates the extensionality of function types. As another example, we show that gradual upcasts are pure functions and, dually, gradual downcasts are strict functions. We show the consistency and applicability of our axiomatic theory by proving that a contract-based implementation using the lazy cast semantics gives a logical relations model of our type theory, where equivalence in GTT implies contextual equivalence of the programs. Since GTT also axiomatizes the dynamic gradual guarantee, our model also establishes this central theorem of gradual typing. The model is parametrized by the implementation of the dynamic types, and so gives a family of implementations that validate type-based optimization and the gradual guarantee.

CCS Concepts: • **Theory of computation → Axiomatic semantics**; • **Software and its engineering → Functional languages**;

Additional Key Words and Phrases: gradual typing, graduality, call-by-push-value

ACM Reference Format:
Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual Type Theory. *Proc. ACM Program. Lang.* 3, POPL, Article 15 (January 2019), 31 pages. https://doi.org/10.1145/3290328

1 INTRODUCTION

Gradually typed languages are designed to support a mix of dynamically typed and statically typed programming styles and preserve the benefits of each. Dynamically typed code can be written without conforming to a syntactic type discipline, so the programmer can always run their program interactively with minimal work. On the other hand, statically typed code provides mathematically sound reasoning principles that justify type-based refactorings, enable compiler optimizations, and underlie formal software verification. The difficulty is accommodating both of these styles and

Authors' addresses: Max S. New, Northeastern University, USA, maxnew@ccs.neu.edu; Daniel R. Licata, Wesleyan University, USA, dlicata@wesleyan.edu; Amal Ahmed, Northeastern University, USA , Inria Paris, France, amal@ccs.neu.edu.

2475-1421/2019/1-ART15
https://doi.org/10.1145/3290328

their benefits simultaneously: allowing the dynamic and static code to interact without forcing the dynamic code to be statically checked or violating the correctness of type-based reasoning.

The linchpin to the design of a gradually typed language is the semantics of *runtime type casts*. These are runtime checks that ensure that typed reasoning principles are valid by checking types of dynamically typed code at the boundary between static and dynamic typing. For instance, when a statically typed function $f : \mathtt{Num} \to \mathtt{Num}$ is applied to a dynamically typed argument $x : ?$, the language runtime must check if x is a number, and otherwise raise a dynamic type error. A programmer familiar with dynamically typed programming might object that this is overly strong: for instance if f is just a constant function $f = \lambda x : \mathtt{Num}.0$ then why bother checking if x is a number since the body of the program does not seem to depend on it? The reason the value is rejected is because the annotation $x : \mathtt{Num}$ should introduce an assumption that that the programmer, compiler and automated tools can rely on for behavioral reasoning in the body of the function. For instance, if the variable x is guaranteed to only be instantiated with numbers, then the programmer is free to replace 0 with $x - x$ or vice-versa. However, if x can be instantiated with a closure, then $x - x$ will raise a runtime type error while 0 will succeed, violating the programmers intuition about the correctness of refactorings. We can formalize such relationships by *observational equivalence* of programs: the two closures $\lambda x : \mathtt{Num}.0$ and $\lambda x : \mathtt{Num}.x - x$ are indistinguishable to any other program in the language. This is precisely the difference between gradual typing and so-called *optional* typing: in an optionally typed language (Hack, TypeScript, Flow), annotations are checked for consistency but are unreliable to the user, so provide no leverage for reasoning. In a gradually typed language, type annotations should relieve the programmer of the burden of reasoning about incorrect inputs, as long as we are willing to accept that the program as a whole may crash, which is already a possibility in many *effectful* statically typed languages.

However, the dichotomy between gradual and optional typing is not as firm as one might like. There have been many different proposed semantics of run-time type checking: "transient" cast semantics [Vitousek et al. 2017] only checks the head connective of a type (number, function, list, ...), "eager" cast semantics [Herman et al. 2010] checks run-time type information on closures, whereas "lazy" cast semantics [Findler and Felleisen 2002] will always delay a type-check on a function until it is called (and there are other possibilities, see e.g. [Siek et al. 2009; Greenberg 2015]). The extent to which these different semantics have been shown to validate type-based reasoning has been limited to syntactic type soundness and blame soundness theorems. In their strongest form, these theorems say "If t is a closed program of type A then it diverges, or reduces to a runtime error blaming dynamically typed code, or reduces to a value that satisfies A to a certain extent." However, the theorem at this level of generality is quite weak, and justifies almost no program equivalences without more information. Saying that a resulting value satisfies type A might be a strong statement, but in transient semantics constrains only the head connective. The blame soundness theorem might also be quite strong, but depends on the definition of blame, which is part of the operational semantics of the language being defined. We argue that these type soundness theorems are only indirectly expressing the actual desired properties of the gradual language, which are *program equivalences in the typed portion of the code* that are not valid in the dynamically typed portion.

Such program equivalences typically include β-like principles, which arise from computation steps, as well as η *equalities*, which express the uniqueness or universality of certain constructions. The η law of the untyped λ-calculus, which states that any λ-term $M \equiv \lambda x.Mx$, is restricted in a typed language to only hold for terms of function type $M : A \to B$ (λ is the unique/universal way of making an element of the function type). This famously "fails" to hold in call-by-value languages in the presence of effects: if M is a program that prints `"hello"` before returning a function, then M will print *now*, whereas $\lambda x.Mx$ will only print when given an argument. But this can be

accommodated with one further modification: the η law is valid in simple call-by-value languages[1] (e.g. SML) if we have a "value restriction" $V \equiv \lambda x.Vx$. This illustrates that η/extensionality rules must be stated for each type connective, and be sensitive to the effects/evaluation order of the terms involved. For instance, the η principle for the boolean type `Bool` *in call-by-value* is that for any term M with a free variable $x : \texttt{Bool}$, M is equivalent to a term that performs an if statement on x: $M \equiv \texttt{if}\ x(M[\texttt{true}/x])(M[\texttt{false}/x])$. If we have an `if` form that is strongly typed (i.e., errors on non-booleans) then this tells us that it is *safe* to run an if statement on any input of boolean type (in CBN, by contrast an if statement forces a thunk and so is not necessarily safe). In addition, even if our `if` statement does some kind of coercion, this tells us that the term M only cares about whether x is "truthy" or "falsy" and so a client is free to change e.g. one truthy value to a different one without changing behavior. This η principle justifies a number of program optimizations, such as dead-code and common subexpression elimination, and hoisting an if statement outside of the body of a function if it is well-scoped ($\lambda x.\texttt{if}\ y\, M\, N \equiv \texttt{if}\ y\, (\lambda x.M)\, (\lambda x.N)$). Any eager datatype, one whose elimination form is given by pattern matching such as $0, +, 1, \times$, `list`, has a similar η principle which enables similar reasoning, such as proofs by induction. The η principles for lazy types *in call-by-name* support dual behavioral reasoning about lazy functions, records, and streams.

An Axiomatic Approach to Gradual Typing. In this paper, we systematically study questions of program equivalence for a class of gradually typed languages by working in an *axiomatic theory* of gradual program equivalence, a language and logic we call *gradual type theory* (GTT). Gradual type theory is the combination of a language of terms and gradual types with a simple logic for proving program equivalence and *error approximation* (equivalence up to one program erroring when the other does not) results. The logic axiomatizes the equational properties gradual programs should satisfy, and offers a high-level syntax for proving theorems about many languages at once: if a language models gradual type theory, then it satisfies all provable equivalences/approximations. Due to its type-theoretic design, different axioms of program equivalence are easily added or removed. Gradual type theory can be used both to explore language design questions and to verify behavioral properties of specific programs, such as correctness of optimizations and refactorings.

To get off the ground, we take two properties of the gradual language for granted. First, we assume a compositionality property: that any cast from A to B can be factored through the dynamic type ?, i.e., the cast $\langle B \Leftarrow A\rangle t$ is equivalent to first casting up from A to ? and then down to B: $\langle B \Leftarrow ?\rangle\langle ? \Leftarrow A\rangle t$. These casts often have quite different performance characteristics, but should have the same extensional behavior: of the cast semantics presented in Siek et al. [2009], only the partially eager detection strategy violates this principle, and this strategy is not common. The second property we take for granted is that the language satisfies the *dynamic gradual guarantee* [Siek et al. 2015a] ("graduality")—a strong correctness theorem of gradual typing— which constrains how changing type annotations changes behavior. Graduality says that if we change the types in a program to be "more precise"—e.g., by changing from the dynamic type to a more precise type such as integers or functions—the program will either produce the same behavior as the original or raise a dynamic type error. Conversely, if a program does not error and some types are made "less precise" then behavior does not change.

We then study what program equivalences are provable in GTT under various assumptions. Our central application is to study when the β, η equalities are satisfied in a gradually typed language. We approach this problem by a surprising tack: rather than defining the behavior of dynamic type casts and then verifying or invalidating the β and η equalities, we *assume* the language satisfies

[1] This does not hold in languages with some intensional feature of functions such as reference equality. We discuss the applicability of our main results more generally in Section 7.

β and η equality and then show that certain reductions of casts are in fact program equivalence *theorems* deducible from the axioms of GTT.

The cast reductions that we show satisfy all three constraints are those given by the "lazy cast semantics" [Findler and Felleisen 2002; Siek et al. 2009]. As a contrapositive, any gradually typed language for which these reductions are not program equivalences is *not* a model of the axioms of gradual type theory. This mean the language violates either compositionality, the gradual guarantee, or one of the β, η axioms—and in practice, it is usually η.

For instance, a transient semantics, where only the top-level connectives are checked, violates η for strict pairs

$$x : A_1 \times A_2 \vdash (\texttt{let}\ (x_1, x_2) = x; 0) \neq 0$$

because the top-level connectives of A_1 and A_2 are only checked when the pattern match is introduced. As a concrete counterexample to contextual equivalence, let A_1, A_2 all be `String`. Because only the top-level connective is checked, $(0, 1)$ is a valid value of type $\texttt{String} \times \texttt{String}$, but pattern matching on the pair ensures that the two components are checked to be strings, so the left-hand side $\texttt{let}\ (x_1, x_2) = (0, 1); 0 \mapsto \mathfrak{U}$ (raises a type error). On the right-hand side, with no pattern, match a value (0) is returned. This means simple program changes that are valid in a typed language, such as changing a function of two arguments to take a single pair of those arguments, are invalidated by the transient semantics. In summary, transient semantics is "lazier" than the types dictate, catching errors only when the term is inspected.

As a subtler example, in call-by-value "eager cast semantics" the $\beta\eta$ principles for all of the eager datatypes ($0, +, 1, \times$, lists, etc.) will be satisfied, but the η principle for the function type $\to$ is violated: there are values $V : A \to A'$ for which $V \neq \lambda x : A.Vx$. For instance, take an arbitrary function value $V : A \to \texttt{String}$ for some type A, and let $V' = \langle A \to ? \Leftarrow A \to \texttt{String} \rangle V$ be the result of casting it to have a dynamically typed output. Then in eager semantics, the following programs are not equivalent:

$$\lambda x : A.V'x \neq V' : A \to ?$$

We cannot observe any difference between these two programs by applying them to arguments, however, they are distinguished from each other by their behavior when *cast*. Specifically, if we cast both sides to $A \to \texttt{Number}$, then $\langle A \to \texttt{Number} \Leftarrow A \to ? \rangle (\lambda x : A.V'x)$ is a value, but $\langle A \to \texttt{Number} \Leftarrow A \to ? \rangle V'$ reduces to an error because `Number` is incompatible with `String`. However this type error might not correspond to any actual typing violation of the program involved. For one thing, the resulting function might never be executed. Furthermore, in the presence of effects, it may be that the original function $V : A \to \texttt{String}$ never returns a string (because it diverges, raises an exception or invokes a continuation), and so that same value casted to $A \to \texttt{Number}$ might be a perfectly valid inhabitant of that type. In summary the "eager" cast semantics is in fact overly eager: in its effort to find bugs faster than "lazy" semantics it disables the very type-based reasoning that gradual typing should provide.

While criticisms of transient semantics on the basis of type soundness have been made before [Greenman and Felleisen 2018], our development shows that the η principles of types are enough to uniquely determine a cast semantics, and helps clarify the trade-off between eager and lazy semantics of function casts.

Technical Overview of GTT. The gradual type theory developed in this paper unifies our previous work on operational (logical relations) reasoning for gradual typing in a call-by-value setting [New and Ahmed 2018] (which did not consider a proof theory), and on an axiomatic proof theory for gradual typing [New and Licata 2018] in a call-by-name setting (which considered only function and product types, and denotational but not operational models).

In this paper, we develop an axiomatic gradual type theory GTT for a unified language that includes *both* call-by-value/eager types and call-by-name/lazy types (Sections 2, 3), and show that

it is sound for contextual equivalence via a logical relations model (Sections 4, 5, 6). Because the η principles for types play a key role in our approach, it is necessary to work in a setting where we can have η principles for both eager and lazy types. We use Levy's Call-by-Push-Value [Levy 2003] (CBPV), which fully and faithfully embeds both call-by-value and call-by-name evaluation with both eager and lazy datatypes,[2] and underlies much recent work on reasoning about effectful programs [Bauer and Pretnar 2013; Lindley et al. 2017]. GTT can prove results in and about existing call-by-value gradually typed languages, and also suggests a design for call-by-name and full call-by-push-value gradually typed languages.

In the prior work [New and Licata 2018; New and Ahmed 2018], gradual type casts are decomposed into upcasts and downcasts, as suggested above. A *type dynamism* relation (corresponding to type precision [Siek et al. 2015a] and naïve subtyping [Wadler and Findler 2009]) controls which casts exist: a type dynamism $A \sqsubseteq A'$ induces an upcast from A to A' and a downcast from A' to A. Then, a *term dynamism* judgement is used for equational/approximational reasoning about programs. Term dynamism relates two terms whose types are related by type dynamism, and the upcasts and downcasts are each *specified* by certain term dynamism judgements holding. This specification axiomatizes only the properties of casts needed to ensure the graduality theorem, and not their precise behavior, so cast reductions can be *proved from it*, rather than stipulated in advance. The specification defines the casts "uniquely up to equivalence", which means that any two implementations satisfying it are behaviorally equivalent.

We generalize this axiomatic approach to call-by-push-value (Section 2), where there are both eager/value types and lazy/computation types. This is both a subtler question than it might at first seem, and has a surprisingly nice answer: we find that upcasts are naturally associated with eager/value types and downcasts with lazy/computation types, and that the modalities relating values and computations induce the downcasts for eager/value types and upcasts for lazy/computation types. Moreover, this analysis articulates an important behavioral property of casts that was proved operationally for call-by-value in [New and Ahmed 2018] but missed for call-by-name in [New and Licata 2018]: upcasts for eager types and downcasts for lazy types are both "pure" in a suitable sense, which enables more refactorings and program optimizations. In particular, we show that these casts can be taken to be (and are essentially forced to be) "complex values" and "complex stacks" (respectively) in call-by-push-value, which corresponds to a behavioral property of *thunkability* and *linearity* [Munch-Maccagnoni 2014]. We argue in Section 7 that this property is related to blame soundness. Our gradual type theory naturally has two dynamic types, a dynamic eager/value type and a dynamic lazy/computation type, where the former can be thought of as a sum of all possible values, and the latter as a product of all possible behaviors. At the language design level, gradual type theory can be used to prove that, for a variety of eager/value and lazy/computation types, the "lazy" semantics of casts is the unique implementation satisfying β, η and graduality (Section 3). These behavioral equivalences can then be used in reasoning about optimizations, refactorings, and correctness of specific programs.

Contract-Based Models. To show the consistency of GTT as a theory, and to give a concrete operational interpretation of its axioms and rules, we provide a concrete model based on an operational semantics. The model is a *contract* interpretation of GTT in that the "built-in" casts of GTT are translated to ordinary functions in a CBPV language that perform the necessary checks.

To keep the proofs high-level, we break the proof into two steps. First (Sections 4, 5), we translate the axiomatic theory of GTT into an axiomatic theory of CBPV extended with recursive types and an uncatchable error, implementing casts by CBPV code that does contract checking. Then (Section 6) we give an operational semantics for the extended CBPV and define a step-indexed

[2]The distinction between "lazy" vs "eager" casts above is different than lazy vs. eager datatypes.

biorthogonal logical relation that interprets the ordering relation on terms as contextual error approximation, which underlies the definition of graduality as presented in [New and Ahmed 2018]. Combining these theorems gives an implementation of the term language of GTT in which β, η are observational equivalences and the dynamic gradual guarantee is satisfied.

Due to the uniqueness theorems of GTT, the only part of this translation that is not predetermined is the definition of the dynamic types themselves and the casts between "ground" types and the dynamic types. We use CBPV to explore the design space of possible implementations of the dynamic types, and give one that faithfully distinguishes all types of GTT, and another more Scheme-like implementation that implements sums and lazy pairs by tag bits. Both can be restricted to the CBV or CBN subsets of CBPV, but the unrestricted variant is actually more faithful to Scheme-like dynamically typed programming, because it accounts for variable-argument functions. Our modular proof architecture allows us to easily prove correctness of β, η and graduality for all of these interpretations.

Contributions. The main contributions of the paper are as follows.

(1) We present Gradual Type Theory in Section 2, a simple axiomatic theory of gradual typing. The theory axiomatizes three simple assumptions about a gradual language: compositionality, graduality, and type-based reasoning in the form of η equivalences.
(2) We prove many theorems in the formal logic of Gradual Type Theory in Section 3. These include the unique implementation theorems for casts, which show that for each type connective of GTT, the η principle for the type ensures that the casts must implement the lazy contract semantics. Furthermore, we show that upcasts are always pure functions and dually that downcasts are always strict functions, as long as the base type casts are pure/strict.
(3) To substantiate that GTT is a reasonable axiomatic theory for gradual typing, we construct *models* of GTT in Sections 4, 5 and 6.3. This proceeds in two stages. First (Section 4), we use call-by-push-value as a typed metalanguage to construct several models of GTT using different recursive types to implement the dynamic types of GTT and interpret the casts as embedding-projection pairs. This extends standard translations of dynamic typing into static typing using type tags: the dynamic value type is constructed as a recursive sum of basic value types, but dually the dynamic computation type is constructed as a recursive *product* of basic computation types. This dynamic computation type naturally models stack-based implementations of variable-arity functions as used in the Scheme language.
(4) We then give an operational model of the term dynamism ordering as contextual error approximation in Sections 5 and 6.3. To construct this model, we extend previous work on logical relations for error approximation from call-by-value to call-by-push-value [New and Ahmed 2018], simplifying the presentation in the process.

Extended version: An extended version of the paper, which includes the omitted cases of definitions, lemmas, and proofs is available in New et al. [2018].

2 AXIOMATIC GRADUAL TYPE THEORY

In this section we introduce the syntax of Gradual Type Theory, an extension of Call-by-push-value [Levy 2003] to support the constructions of gradual typing. First we introduce call-by-push-value and then describe in turn the gradual typing features: dynamic types, casts, and the dynamism orderings on types and terms.

2.1 Background: Call-by-Push-Value

GTT is an extension of CBPV, so we first present CBPV as the unshaded rules in Figure 1. CBPV makes a distinction between *value types* A and *computation types* $\underline{B}$, where value types classify

$$
\begin{array}{lll}
A ::= & ? \mid UB \mid 0 \mid A_1 + A_2 \mid 1 \mid A_1 \times A_2 & \\
V ::= & \langle A' \leftsquigarrow A \rangle V \mid x \mid \mathtt{abort}\ V & \\
& \mid \mathtt{inl}\ V \mid \mathtt{inr}\ V & \\
& \mid \mathtt{case}\ V\{x_1.V_1 \mid x_2.V_2\} & \\
& \mid () \mid \mathtt{split}\ V\ \mathtt{to}\ ().V' & \\
& \mid (V_1, V_2) \mid \mathtt{split}\ V\ \mathtt{to}\ (x,y).V' & \\
& \mid \mathtt{thunk}\ M & \\
\Gamma ::= & \cdot \mid \Gamma, x : A & \\
\Phi ::= & \cdot \mid \Phi, x \sqsubseteq x' : A \sqsubseteq A' &
\end{array}
\qquad
\begin{array}{lll}
\underline{B} ::= & ¿ \mid \underline{F}A \mid \top \mid \underline{B}_1 \,\&\, \underline{B}_2 \mid A \to \underline{B} & \\
M, S ::= & \langle \underline{B} \nleftarrow \underline{B}' \rangle M \mid \bullet \mid \mathfrak{V}_{\underline{B}} & \\
& \mid \mathtt{abort}\ V \mid \mathtt{case}\ V\{x_1.M_1 \mid x_2.M_2\} & \\
& \mid \mathtt{split}\ V\ \mathtt{to}\ ().M \mid \mathtt{split}\ V\ \mathtt{to}\ (x,y).M & \\
& \mid \mathtt{force}\ V \mid \mathtt{ret}\ V \mid \mathtt{bind}\ x \leftarrow M; N & \\
& \mid \lambda x : A.M \mid M\,V & \\
& \mid \{\} \mid \{\pi \mapsto M_1 \mid \pi' \mapsto M_2\} & \\
& \mid \pi M \mid \pi' M & \\
\Delta ::= & \cdot \mid \bullet : \underline{B} & \\
\Psi ::= & \cdot \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' &
\end{array}
$$

$$
\boxed{\Gamma \vdash V : A \text{ and } \Gamma \mid \Delta \vdash M : \underline{B}}
$$

$$
\textsc{UpCast}\ \frac{\Gamma \vdash V : A \qquad A \sqsubseteq A'}{\Gamma \vdash \langle A' \leftsquigarrow A \rangle V : A'}
\qquad
\textsc{DnCast}\ \frac{\Gamma \mid \Delta \vdash M : \underline{B}' \qquad \underline{B} \sqsubseteq \underline{B}'}{\Gamma \mid \Delta \vdash \langle \underline{B} \nleftarrow \underline{B}' \rangle M : \underline{B}}
$$

$$
\textsc{Var}\ \frac{}{\Gamma, x : A, \Gamma' \vdash x : A}
\qquad
\textsc{Hole}\ \frac{}{\Gamma \mid \bullet : \underline{B} \vdash \bullet : \underline{B}}
\qquad
\textsc{Err}\ \frac{}{\Gamma \mid \cdot \vdash \mathfrak{V}_{\underline{B}} : \underline{B}}
$$

$$
1\mathrm{I}\ \frac{}{\Gamma \vdash () : 1}
\qquad
1\mathrm{E}\ \frac{\Gamma \vdash V : 1 \qquad \Gamma \mid \Delta \vdash E : T}{\Gamma \mid \Delta \vdash \mathtt{split}\ V\ \mathtt{to}\ ().E : T}
\qquad
\times\mathrm{I}\ \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2}
\qquad
\times\mathrm{E}\ \frac{\Gamma \vdash V : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \mid \Delta \vdash E : T}{\Gamma \mid \Delta \vdash \mathtt{split}\ V\ \mathtt{to}\ (x,y).E : T}
$$

$$
U\mathrm{I}\ \frac{\Gamma \mid \cdot \vdash M : \underline{B}}{\Gamma \vdash \mathtt{thunk}\ M : U\underline{B}}
\qquad
U\mathrm{E}\ \frac{\Gamma \vdash V : U\underline{B}}{\Gamma \mid \cdot \vdash \mathtt{force}\ V : \underline{B}}
\qquad
F\mathrm{I}\ \frac{\Gamma \vdash V : A}{\Gamma \mid \cdot \vdash \mathtt{ret}\ V : \underline{F}A}
\qquad
F\mathrm{E}\ \frac{\Gamma \mid \Delta \vdash M : \underline{F}A \qquad \Gamma, x : A \mid \cdot \vdash N : \underline{B}}{\Gamma \mid \Delta \vdash \mathtt{bind}\ x \leftarrow M; N : \underline{B}}
$$

$$
\to\mathrm{I}\ \frac{\Gamma, x : A \mid \Delta \vdash M : \underline{B}}{\Gamma \mid \Delta \vdash \lambda x : A.M : A \to \underline{B}}
\qquad
\to\mathrm{E}\ \frac{\Gamma \mid \Delta \vdash M : A \to \underline{B} \qquad \Gamma \vdash V : A}{\Gamma \mid \Delta \vdash M\,V : \underline{B}}
$$

Fig. 1. GTT Syntax and Term Typing (+ and & typing rules in extended version)

values $\Gamma \vdash V : A$ and computation types classify *computations* $\Gamma \vdash M : \underline{B}$. Effects are computations: for example, we might have an error computation $\mathfrak{V}_{\underline{B}} : \underline{B}$ of every computation type, or printing $\mathtt{print}\ V; M : \underline{B}$ if $V : \mathtt{string}$ and $M : \underline{B}$, which prints V and then behaves as M.

Value types and complex values. The value types include *eager* products 1 and $A_1 \times A_2$ and sums 0 and $A_1 + A_2$, which behave as in a call-by-value/eager language (e.g. a pair is only a value when its components are). The notion of value V is more permissive than one might expect, and expressions $\Gamma \vdash V : A$ are sometimes called *complex values* to emphasize this point: complex values include not only closed runtime values, but also open values that have free value variables (e.g. $x : A_1, x_2 : A_2 \vdash (x_1, x_2) : A_1 \times A_2$), and expressions that pattern-match on values (e.g. $p : A_1 \times A_2 \vdash \mathtt{split}\ p\ \mathtt{to}\ (x_1, x_2).(x_2, x_1) : A_2 \times A_1$). Thus, the complex values $x : A \vdash V : A'$ are a syntactic class of "pure functions" from A to A' (though there is no pure function *type* internalizing this judgement), which can be treated like values by a compiler because they have no effects (e.g. they can be dead-code-eliminated, common-subexpression-eliminated, and so on). For each pattern-matching construct (e.g. case analysis on a sum, splitting a pair), we have both an elimination rule whose branches are values (e.g. $\mathtt{split}\ p\ \mathtt{to}\ (x_1, x_2).V$) and one whose branches are computations (e.g. $\mathtt{split}\ p\ \mathtt{to}\ (x_1, x_2).M$). To abbreviate the typing rules for both in Figure 1,

we use the following convention: we write $E ::= V \mid M$ for either a complex value or a computation, and $T ::= A \mid \underline{B}$ for either a value type A or a computation type $\underline{B}$, and a judgement $\Gamma \mid \Delta \vdash E : T$ for either $\Gamma \vdash V : A$ or $\Gamma \mid \Delta \vdash M : \underline{B}$ (this is a bit of an abuse of notation because Δ is not present in the former). Complex values can be translated away without loss of expressiveness by moving all pattern-matching into computations (see Section 5), at the expense of using a behavioral condition of *thunkability* [Munch-Maccagnoni 2014] to capture the properties complex values have (for example, an analogue of $\texttt{let}\ x = V; \texttt{let}\ x' = V'; M \equiv \texttt{let}\ x' = V'; \texttt{let}\ x = V; M$ — complex values can be reordered, while arbitrary computations cannot).

Shifts. A key notion in CBPV is the *shift* types $\underline{F}A$ and $U\underline{B}$, which mediate between value and computation types: $\underline{F}A$ is the computation type of potentially effectful programs that return a value of type A, while $U\underline{B}$ is the value type of thunked computations of type $\underline{B}$. The introduction rule for $\underline{F}A$ is returning a value of type A ($\texttt{ret}\ \mathrm{V}$), while the elimination rule is sequencing a computation $M : \underline{F}A$ with a computation $x : A \vdash N : \underline{B}$ to produce a computation of a $\underline{B}$ ($\texttt{bind}\ x \leftarrow M; N$). While any closed complex value V is equivalent to an actual value, a computation of type $\underline{F}A$ might perform effects (e.g. printing) before returning a value, or might error or non-terminate and not return a value at all. The introduction and elimination rules for U are written $\texttt{thunk}\ M$ and $\texttt{force}\ V$, and say that computations of type $\underline{B}$ are bijective with values of type $U\underline{B}$. As an example of the action of the shifts, 1 is the trivial value type, so $\underline{F}1$ is the type of computations that can perform effects with the possibility of terminating successfully by returning $()$, and $U\underline{F}1$ is the value type where such computations are delayed values.

Computation types. The computation type constructors in CBPV include lazy unit/products $\top$ and $\underline{B}_1 \ \& \ \underline{B}_2$, which behave as in a call-by-name/lazy language (e.g. a component of a lazy pair is evaluated only when it is projected). Functions $A \to \underline{B}$ have a value type as input and a computation type as a result. The equational theory of effects in CBPV computations may be surprising to those familiar only with call-by-value, because at higher computation types effects have a call-by-name-like equational theory. For example, at computation type $A \to \underline{B}$, we have an equality $\texttt{print}\ c; \lambda x.M = \lambda x.\texttt{print}\ c; M$. Intuitively, the reason is that $A \to \underline{B}$ is not treated as an *observable* type (one where computations are run): the states of the operational semantics are only those computations of type $\underline{F}A$ for some value type A. Thus, "running" a function computation means supplying it with an argument, and applying both of the above to an argument V is defined to result in $\texttt{print}\ c; M[V/x]$. This does *not* imply that the corresponding equations holds for the call-by-value function type, which we discuss below.

Complex stacks. Just as the complex values V are a syntactic class terms that have no effects, CBPV includes a judgement for "stacks" S, a syntactic class of terms that reflect *all* effects of their input. A *stack* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{B}'$ can be thought of as a linear/strict function from $\underline{B}$ to $\underline{B}'$, which *must* use its input hole $\bullet$ *exactly* once at the head redex position. Consequently, effects can be hoisted out of stacks, because we know the stack will run them exactly once and first. For example, there will be contextual equivalences $S[\mho/\bullet] = \mho$ and $S[\texttt{print}\ V; M] = \texttt{print}\ V; S[M/\bullet]$. Just as complex values include pattern-matching, *complex stacks* include pattern-matching on values and introduction forms for the stack's output type. For example, $\bullet : \underline{B}_1 \ \& \ \underline{B}_2 \vdash \{\pi \mapsto \pi' \bullet \mid \pi' \mapsto \pi \bullet\} : \underline{B}_2 \ \& \ \underline{B}_1$ is a complex stack, even though it mentions $\bullet$ more than once, because running it requires choosing a projection to get to an observable of type $\underline{F}A$, so *each time it is run* it uses $\bullet$ exactly once. In the equational theory of CBPV, $\underline{F}$ and U are *adjoint*, in the sense that stacks $\bullet : \underline{F}A \vdash S : \underline{B}$ are bijective with values $x : A \vdash V : U\underline{B}$, as both are bijective with computations $x : A \vdash M : \underline{B}$.

To compress the presentation in Figure 1, we use a typing judgement $\Gamma \mid \Delta \vdash M : \underline{B}$ with a "stoup", a typing context Δ that is either empty or contains exactly one assumption $\bullet : \underline{B}$, so $\Gamma \mid \cdot \vdash M : \underline{B}$ is a computation, while $\Gamma \mid \bullet : \underline{B} \vdash M : \underline{B}'$ is a stack. The (omitted) typing rules for $\top$ and $\&$ treat the stoup additively (it is arbitrary in the conclusion and the same in all premises); for a function

application to be a stack, the stack input must occur in the function position. The elimination form for $U\underline{B}$, `force` V, is the prototypical non-stack computation (Δ is required to be empty), because forcing a thunk does not use the stack's input.

Embedding call-by-value and call-by-name. To translate call-by-value (CBV) into CBPV, a judgement $x_1 : A_1, \ldots, x_n : A_n \vdash e : A$ is interpreted as a computation $x_1 : A_1^v, \ldots, x_n : A_n^v \vdash e^v : \underline{F}A^v$, where call-by-value products and sums are interpreted as $\times$ and $+$, and the call-by-value function type $A \to A'$ as $U(A^v \to \underline{F}A'^v)$. Thus, a call-by-value term $e : A \to A'$, which should mean an effectful computation of a function value, is translated to a computation $e^v : \underline{F}U(A^v \to \underline{F}A'^v)$. Here, the comonad $\underline{F}U$ offers an opportunity to perform effects *before* returning a function value—so under translation the CBV terms `print` c; $\lambda x.e$ and $\lambda x.$`print` c; e will not be contextually equivalent. To translate call-by-name (CBN) to CBPV, a judgement $x_1 : \underline{B}_1, \ldots, x_m : \underline{B}_m \vdash e : \underline{B}$ is translated to $x_1 : U\underline{B}_1{}^n, \ldots, x_m : U\underline{B}_m{}^n \vdash e^n : \underline{B}^n$, representing the fact that call-by-name terms are passed thunked arguments. Product types are translated to $\top$ and $\times$, while a CBN function $B \to B'$ is translated to $U\underline{B}^n \to \underline{B}'^n$ with a thunked argument. Sums $B_1 + B_2$ are translated to $\underline{F}(U\underline{B_1}{}^n + U\underline{B_2}{}^n)$, making the "lifting" in lazy sums explicit. Call-by-push-value *subsumes* call-by-value and call-by-name in that these embeddings are *full and faithful*: two CBV or CBN programs are equivalent if and only if their embeddings into CBPV are equivalent, and every CBPV program with a CBV or CBN type can be back-translated.

Extensionality/η Principles. The main advantage of CBPV for our purposes is that it accounts for the η/extensionality principles of both eager/value and lazy/computation types, because value types have η principles relating them to the value assumptions in the context Γ, while computation types have η principles relating them to the result type of a computation $\underline{B}$. For example, the η principle for sums says that any complex value or computation $x : A_1 + A_2 \vdash E : T$ is equivalent to `case` $x\{x_1.E[\texttt{inl}\ x_1/x] \mid x_2.E[\texttt{inr}\ x_2/x]\}$, i.e. a case on a value can be moved to any point in a program (where all variables are in scope) in an optimization. Given this, the above translations of CBV and CBN into CBPV explain why η for sums holds in CBV but not CBN: in CBV, $x : A_1 + A_2 \vdash E : T$ is translated to a term with $x : A_1 + A_2$ free, but in CBN, $x : B_1 + B_2 \vdash E : T$ is translated to a term with $x : U\underline{F}(U\underline{B}_1 + U\underline{B}_2)$ free, and the type $U\underline{F}(U\underline{B}_1 + U\underline{B}_2)$ of monadic computations that return a sum does not satisfy the η principle for sums in CBPV. Dually, the η principle for functions in CBPV is that any computation $M : A \to \underline{B}$ is equal to $\lambda x.M\,x$. A CBN term $e : B \to B'$ is translated to a CBPV computation of type $U\underline{B} \to \underline{B}'$, to which CBPV function extensionality applies, while a CBV term $e : A \to A'$ is translated to a computation of type $\underline{F}U(A \to \underline{F}A')$, which does not satisfy the η rule for functions. We discuss a formal statement of these η principles with term dynamism below.

2.2 The Dynamic Type(s)

Next, we discuss the additions that make CBPV into our gradual type theory GTT. A dynamic type plays a key role in gradual typing, and since GTT has two different kinds of types, we have a new question of whether the dynamic type should be a value type, or a computation type, or whether we should have *both* a dynamic value type and a dynamic computation type. Our modular, type-theoretic presentation of gradual typing allows us to easily explore these options, though we find that having both a dynamic value ? and a dynamic computation type $\underline{?}$ gives the most natural implementation (see Section 4.2). Thus, we add both ? and $\underline{?}$ to the grammar of types in Figure 1. We do *not* give introduction and elimination rules for the dynamic types, because we would like constructions in GTT to imply results for many different possible implementations of them. Instead, the terms for the dynamic types will arise from type dynamism and casts.

$\boxed{A \sqsubseteq A' \text{ and } \underline{B} \sqsubseteq \underline{B}'}$

$$\textsc{VTyRefl}\ \frac{}{A \sqsubseteq A} \qquad \textsc{VTyTrans}\ \frac{A \sqsubseteq A' \qquad A' \sqsubseteq A''}{A \sqsubseteq A''} \qquad \textsc{CTyRefl}\ \frac{}{\underline{B} \sqsubseteq \underline{B}} \qquad \textsc{CTyTrans}\ \frac{\underline{B} \sqsubseteq \underline{B}' \qquad \underline{B}' \sqsubseteq \underline{B}''}{\underline{B} \sqsubseteq \underline{B}''}$$

$$\textsc{VTyTop}\ \frac{}{A \sqsubseteq ?} \qquad U\textsc{Mon}\ \frac{\underline{B} \sqsubseteq \underline{B}'}{U\underline{B} \sqsubseteq U\underline{B}'} \qquad +\textsc{Mon}\ \frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 + A_2 \sqsubseteq A_1' + A_2'} \qquad \times\textsc{Mon}\ \frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 \times A_2 \sqsubseteq A_1' \times A_2'}$$

$$\textsc{CTyTop}\ \frac{}{\underline{B} \sqsubseteq \underline{¿}} \qquad F\textsc{Mon}\ \frac{A \sqsubseteq A'}{F A \sqsubseteq F A'} \qquad \&\textsc{Mon}\ \frac{\underline{B}_1 \sqsubseteq \underline{B}_1' \qquad \underline{B}_2 \sqsubseteq \underline{B}_2'}{\underline{B}_1 \,\&\, \underline{B}_2 \sqsubseteq \underline{B}_1' \,\&\, \underline{B}_2'} \qquad \to\textsc{Mon}\ \frac{A \sqsubseteq A' \qquad \underline{B} \sqsubseteq \underline{B}'}{A \to \underline{B} \sqsubseteq A' \to \underline{B}'}$$

Fig. 2. GTT Type Dynamism

2.3 Type Dynamism

The *type dynamism* relation of gradual type theory is written $A \sqsubseteq A'$ and read as "A is less dynamic than A'"; intuitively, this means that A' supports more behaviors than A. Our previous work [New and Ahmed 2018; New and Licata 2018] analyzes this as the existence of an *upcast* from A to A' and a downcast from A' to A which form an embedding-projection pair (*ep pair*) for term error approximation (an ordering where runtime errors are minimal): the upcast followed by the downcast is a no-op, while the downcast followed by the upcast might error more than the original term, because it imposes a run-time type check. Syntactically, type dynamism is defined (1) to be reflexive and transitive (a preorder), (2) where every type constructor is monotone in all positions, and (3) where the dynamic type is greatest in the type dynamism ordering. This last condition, *the dynamic type is the most dynamic type*, implies the existence of an upcast $\langle ? \nwarrow A \rangle$ and a downcast $\langle A \swarrow ? \rangle$ for every type A: any type can be embedded into it and projected from it. However, this by design does not characterize ? uniquely—instead, it is open-ended exactly which types exist (so that we can always add more), and some properties of the casts are undetermined; we exploit this freedom in Section 4.2.

This extends in a straightforward way to CBPV's distinction between value and computation types in Figure 2: there is a type dynamism relation for value types $A \sqsubseteq A'$ and for computation types $\underline{B} \sqsubseteq \underline{B}'$, which (1) each are preorders (VTyRefl, VTyTrans, CTyRefl, CTyTrans), (2) every type constructor is monotone (+Mon, ×Mon, &Mon ,→Mon) where the shifts $\underline{F}$ and U switch which relation is being considered (UMon, FMon), and (3) the dynamic types ? and $\underline{¿}$ are the most dynamic value and computation types respectively (VTyTop, CTyTop). For example, we have $U(A \to \underline{F}A') \sqsubseteq U(? \to \underline{F}?)$, which is the analogue of $A \to A' \sqsubseteq ? \to ?$ in call-by-value: because $\to$ preserves embedding-retraction pairs, it is monotone, not contravariant, in the domain [New and Ahmed 2018; New and Licata 2018].

2.4 Casts

It is not immediately obvious how to add type casts to CPBV, because CBPV exposes finer judgemental distinctions than previous work considered. However, we can arrive at a first proposal by considering how previous work would be embedded into CBPV. In the previous work on both CBV and CBN [New and Ahmed 2018; New and Licata 2018] every type dynamism judgement $A \sqsubseteq A'$ induces both an upcast from A to A' and a downcast from A' to A. Because CBV types are associated to CBPV value types and CBN types are associated to CBPV computation types, this suggests that each value type dynamism $A \sqsubseteq A'$ should induce an upcast and a downcast, and

each computation type dynamism $\underline{B} \sqsubseteq \underline{B}'$ should also induce an upcast and a downcast. In CBV, a cast from A to A' typically can be represented by a CBV function $A \to A'$, whose analogue in CBPV is $U(A \to \underline{F}A')$, and values of this type are bijective with computations $x : A \vdash M : \underline{F}A'$, and further with stacks $\bullet : \underline{F}A \vdash S : \underline{F}A'$. This suggests that a *value* type dynamism $A \sqsubseteq A'$ should induce an embedding-projection pair of *stacks* $\bullet : \underline{F}A \vdash S_u : \underline{F}A'$ and $\bullet : \underline{F}A' \vdash S_d : \underline{F}A$, which allow both the upcast and downcast to a priori be effectful computations. Dually, a CBN cast typically can be represented by a CBN function of type $B \to B'$, whose CBPV analogue is a computation of type $U\underline{B} \to \underline{B}'$, which is equivalent with a computation $x : U\underline{B} \vdash M : \underline{B}'$, and with a value $x : U\underline{B} \vdash V : U\underline{B}'$. This suggests that a *computation* type dynamism $\underline{B} \sqsubseteq \underline{B}'$ should induce an embedding-projection pair of *values* $x : U\underline{B} \vdash V_u : U\underline{B}'$ and $x : U\underline{B}' \vdash V_d : U\underline{B}$, where both the upcast and the downcast again may a priori be (co)effectful, in the sense that they may not reflect all effects of their input.

However, this analysis ignores an important property of CBV casts in practice: *upcasts* always terminate without performing any effects, and in some systems upcasts are even defined to be values, while only the *downcasts* are effectful (introduce errors). For example, for many types A, the upcast from A to ? is an injection into a sum/recursive type, which is a value constructor. Our previous work on a logical relation for call-by-value gradual typing [New and Ahmed 2018] proved that all upcasts were pure in this sense as a consequence of the embedding-projection pair properties (but their proof depended on the only effects being divergence and type error). In GTT, we can make this property explicit in the syntax of the casts, by making the upcast $\langle A' \nwarrow A \rangle$ induced by a value type dynamism $A \sqsubseteq A'$ itself a complex value, rather than computation. On the other hand, many downcasts between value types are implemented as a case-analysis looking for a specific tag and erroring otherwise, and so are not complex values.

We can also make a dual observation about CBN casts. The *downcast* arising from $\underline{B} \sqsubseteq \underline{B}'$ has a stronger property than being a computation $x : U\underline{B}' \vdash M : \underline{B}$ as suggested above: it can be taken to be a stack $\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet : \underline{B}$, because a downcasted computation evaluates the computation it is "wrapping" exactly once. One intuitive justification for this point of view, which we make precise in Section 4, is to think of the dynamic computation type $\underline{?}$ as a recursive *product* of all possible behaviors that a computation might have, and the downcast as a recursive type unrolling and product projection, which is a stack. From this point of view, an *upcast* can introduce errors, because the upcast of an object supporting some "methods" to one with all possible methods will error dynamically on the unimplemented ones.

These observations are expressed in the (shaded) UpCast and DnCasts rules for casts in Figure 1: the upcast for a value type dynamism is a complex value, while the downcast for a computation type dynamism is a stack (if its argument is). Indeed, this description of casts is simpler than the intuition we began the section with: rather than putting in both upcasts and downcasts for all value and computation type dynamisms, it suffices to put in only *upcasts* for *value* type dynamisms and *downcasts* for *computation* type dynamisms, because of monotonicity of type dynamism for $U/\underline{F}$ types. The *downcast* for a *value* type dynamism $A \sqsubseteq A'$, as a stack $\bullet : \underline{F}A' \vdash \langle \underline{F}A \swarrow \underline{F}A' \rangle \bullet : \underline{F}A$ as described above, is obtained from $\underline{F}A \sqsubseteq \underline{F}A'$ as computation types. The upcast for a computation type dynamism $\underline{B} \sqsubseteq \underline{B}'$ as a value $x : U\underline{B} \vdash \langle U\underline{B}' \nwarrow U\underline{B} \rangle x : U\underline{B}'$ is obtained from $U\underline{B} \sqsubseteq U\underline{B}'$ as value types. Moreover, we will show below that the value upcast $\langle A' \nwarrow A \rangle$ induces a stack $\bullet : \underline{F}A \vdash \ldots : \underline{F}A'$ that behaves like an upcast, and dually for the downcast, so this formulation implies the original formulation above.

We justify this design in two ways in the remainder of the paper. In Section 4, we show how to implement casts by a contract translation to CBPV where upcasts are complex values and downcasts are complex stacks. However, one goal of GTT is to be able to prove things about many gradually

$$\boxed{\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A' \text{ and } \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}$$

$$\textsc{TmDynRefl}\ \frac{}{\Gamma \sqsubseteq \Gamma \mid \Delta \sqsubseteq \Delta \vdash E \sqsubseteq E : T \sqsubseteq T} \qquad \textsc{TmDynVar}\ \frac{}{\Phi, x \sqsubseteq x' : A \sqsubseteq A', \Phi' \vdash x \sqsubseteq x' : A \sqsubseteq A'}$$

$$\textsc{TmDynTrans}\ \frac{\Gamma \sqsubseteq \Gamma' \mid \Delta \sqsubseteq \Delta' \vdash E \sqsubseteq E' : T \sqsubseteq T' \quad \Gamma' \sqsubseteq \Gamma'' \mid \Delta' \sqsubseteq \Delta'' \vdash E' \sqsubseteq E'' : T' \sqsubseteq T''}{\Gamma \sqsubseteq \Gamma'' \mid \Delta \sqsubseteq \Delta'' \vdash E \sqsubseteq E'' : T \sqsubseteq T''}$$

$$\textsc{TmDynValSubst}\ \frac{\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A' \quad \Phi, x \sqsubseteq x' : A \sqsubseteq A', \Phi' \mid \Psi \vdash E \sqsubseteq E' : T \sqsubseteq T'}{\Phi \mid \Psi \vdash E[V/x] \sqsubseteq E'[V'/x'] : T \sqsubseteq T'}$$

$$\textsc{TmDynHole}\ \frac{}{\Phi \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'}$$

$$\textsc{TmDynStkSubst}\ \frac{\Phi \mid \Psi \vdash M_1 \sqsubseteq M_1' : \underline{B}_1 \sqsubseteq \underline{B}_1' \quad \Phi \mid \bullet \sqsubseteq \bullet : \underline{B}_1 \sqsubseteq \underline{B}_1' \vdash M_2 \sqsubseteq M_2' : \underline{B}_2 \sqsubseteq \underline{B}_2'}{\Phi \mid \Psi \vdash M_2[M_1/\bullet] \sqsubseteq M_2'[M_1'/\bullet] : \underline{B}_2 \sqsubseteq \underline{B}_2'}$$

$$\times\textsc{ICong}\ \frac{\Phi \vdash V_1 \sqsubseteq V_1' : A_1 \sqsubseteq A_1' \quad \Phi \vdash V_2 \sqsubseteq V_2' : A_2 \sqsubseteq A_2'}{\Phi \vdash (V_1, V_2) \sqsubseteq (V_1', V_2') : A_1 \times A_2 \sqsubseteq A_1' \times A_2'} \qquad \to\textsc{ICong}\ \frac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \mid \Psi \vdash \lambda x : A.M \sqsubseteq \lambda x' : A'.M' : A \to \underline{B} \sqsubseteq A' \to \underline{B}'}$$

$$\times\textsc{ECong}\ \frac{\Phi \vdash V \sqsubseteq V' : A_1 \times A_2 \sqsubseteq A_1' \times A_2' \quad \Phi, x \sqsubseteq x' : A_1 \sqsubseteq A_1', y \sqsubseteq y' : A_2 \sqsubseteq A_2' \mid \Psi \vdash E \sqsubseteq E' : T \sqsubseteq T'}{\Phi \mid \Psi \vdash \mathtt{split}\ V\ \mathtt{to}\ (x,y).E \sqsubseteq \mathtt{split}\ V'\ \mathtt{to}\ (x',y').E' : T \sqsubseteq T'} \qquad \to\textsc{ECong}\ \frac{\Phi \mid \Psi \vdash M \sqsubseteq M' : A \to \underline{B} \sqsubseteq A' \to \underline{B}' \quad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \mid \Psi \vdash M\,V \sqsubseteq M'\,V' : \underline{B} \sqsubseteq \underline{B}'}$$

$$F\textsc{ICong}\ \frac{\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \mid \cdot \vdash \mathtt{ret}\ V \sqsubseteq \mathtt{ret}\ V' : \underline{F}A \sqsubseteq \underline{F}A'} \qquad F\textsc{ECong}\ \frac{\Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{F}A \sqsubseteq \underline{F}A' \quad \Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \cdot \vdash N \sqsubseteq N' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \mid \Psi \vdash \mathtt{bind}\ x \leftarrow M; N \sqsubseteq \mathtt{bind}\ x' \leftarrow M'; N' : \underline{B} \sqsubseteq \underline{B}'}$$

Fig. 3. GTT Term Dynamism (Structural and CongruenceRules) (Rules for U, 1, $+$, 0, $\&$, $\top$ in extended version)

typed languages at once, by giving different models, so one might wonder whether this design rules out useful models of gradual typing where casts can have more general effects. In Theorem 3.10, we show instead that our design choice is forced for all casts, as long as the casts between ground types and the dynamic types are values/stacks.

2.5 Term Dynamism: Judgements and Structural Rules

The final piece of GTT is the *term dynamism* relation, a syntactic judgement that is used for reasoning about the behavioral properties of terms in GTT. To a first approximation, term dynamism can be thought of as syntactic rules for reasoning about *contextual approximation* relative to errors (not divergence), where $E \sqsubseteq E'$ means that either E errors or E and E' have the same result. However, a key idea in GTT is to consider a *heterogeneous* term dynamism judgement $E \sqsubseteq E' : T \sqsubseteq T'$ between terms $E : T$ and $E' : T'$ where $T \sqsubseteq T'$—i.e. relating two terms at two different types, where the type on the right is more dynamic than the type on the right. This judgement structure allows simple axioms characterizing the behavior of casts [New and Licata 2018] and axiomatizes the graduality property [Siek et al. 2015a]. Here, we break this judgement up into value dynamism $V \sqsubseteq V' : A \sqsubseteq A'$ and computation dynamism $M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'$. To support reasoning about open terms, the full form of the judgements are

- $\Gamma \sqsubseteq \Gamma' \vdash V \sqsubseteq V' : A \sqsubseteq A'$ where $\Gamma \vdash V : A$ and $\Gamma' \vdash V' : A'$ and $\Gamma \sqsubseteq \Gamma'$ and $A \sqsubseteq A'$.
- $\Gamma \sqsubseteq \Gamma' \mid \Delta \sqsubseteq \Delta' \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'$ where $\Gamma \mid \Delta \vdash M : \underline{B}$ and $\Gamma' \mid \Delta' \vdash M' : \underline{B}'$.

where $\Gamma \sqsubseteq \Gamma'$ is the pointwise lifting of value type dynamism, and $\Delta \sqsubseteq \Delta'$ is the optional lifting of computation type dynamism. We write $\Phi : \Gamma \sqsubseteq \Gamma'$ and $\Psi : \Delta \sqsubseteq \Delta'$ as syntax for "zipped" pairs of contexts that are pointwise related by type dynamism, $x_1 \sqsubseteq x_1' : A_1 \sqsubseteq A_1', \ldots, x_n \sqsubseteq x_n' : A_n \sqsubseteq A_n'$, which correctly suggests that one can substitute related terms for related variables. We will implicitly zip/unzip pairs of contexts, and sometimes write e.g. $\Gamma \sqsubseteq \Gamma$ to mean $x \sqsubseteq x : A \sqsubseteq A$ for all $x : A$ in Γ.

The main point of our rules for term dynamism is that *there are no type-specific axioms in the definition* beyond the $\beta\eta$-axioms that the type satisfies in a non-gradual language. Thus, adding a new type to gradual type theory does not require any a priori consideration of its gradual behavior in the language definition; instead, this is deduced as a theorem in the type theory. The basic structural rules of term dynamism in Figure 3 say that it is reflexive and transitive (TmDynRefl, TmDynTrans), that assumptions can be used and substituted for (TmDynVar, TmDynValSubst, TmDynHole, TmDynStkSubst), and that every term constructor is monotone (the Cong rules).

We will often abbreviate a "homogeneous" term dynamism (where the type or context dynamism is given by reflexivity) by writing e.g. $\Gamma \vdash V \sqsubseteq V' : A \sqsubseteq A'$ for $\Gamma \sqsubseteq \Gamma \vdash V \sqsubseteq V' : A \sqsubseteq A'$, or $\Phi \vdash V \sqsubseteq V' : A$ for $\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A$, and similarly for computations. The entirely homogeneous judgements $\Gamma \vdash V \sqsubseteq V' : A$ and $\Gamma \mid \Delta \vdash M \sqsubseteq M' : \underline{B}$ can be thought of as a syntax for contextual error approximation (as we prove below). We write $V \sqsupseteq\sqsubseteq V'$ ("equidynamism") to mean term dynamism relations in both directions (which requires that the types are also equidynamic $\Gamma \sqsupseteq\sqsubseteq \Gamma'$ and $A \sqsubseteq A'$), which is a syntactic judgement for contextual equivalence.

2.6 Term Dynamism: Axioms

Finally, we assert some term dynamism axioms that describe the behavior of programs. The cast universal properties at the top of Figure 4, following New and Licata [2018], say that the defining property of an upcast from A to A' is that it is the least dynamic term of type A' that is more dynamic that x, a "least upper bound". That is, $\langle A' \nwarrow A \rangle x$ is a term of type A' that is more dynamic that x (the "bound" rule), and for any other term x' of type A' that is more dynamic than x, $\langle A' \nwarrow A \rangle x$ is less dynamic than x' (the "best" rule). Dually, the downcast $\langle \underline{B} \swarrow \underline{B}' \rangle \bullet$ is the most dynamic term of type $\underline{B}$ that is less dynamic than $\bullet$, a "greatest lower bound". These defining properties are entirely independent of the types involved in the casts, and do not change as we add or remove types from the system.

We will show that these defining properties already imply that the shift of the upcast $\langle A' \nwarrow A \rangle$ forms a Galois connection/adjunction with the downcast $\langle \underline{F}A \swarrow \underline{F}A' \rangle$, and dually for computation types (see Theorem 3.3). They do not automatically form a Galois insertion/coreflection/embedding-projection pair, but we can add this by the retract axioms in Figure 4. Together with other theorems of GTT, these axioms imply that any upcast followed by its corresponding downcast is the identity (see Theorem 3.4). This specification of casts leaves some behavior undefined: for example, we cannot prove in the theory that $\langle \underline{F}1 + 1 \swarrow \underline{F}? \rangle \langle ? \nwarrow 1 \rangle$ reduces to an error. We choose this design because there are valid models in which it is not an error, for instance if the unique value of 1 is represented as the boolean `true`. In Section 4.2, we show additional axioms that fully characterize the behavior of the dynamic type.

The type universal properties in the middle of the figure, which are taken directly from CBPV, assert the $\beta\eta$ rules for each type as (homogeneous) term equidynamisms—these should be understood as having, as implicit premises, the typing conditions that make both sides type check, in equidynamic contexts.

The final axioms assert properties of the run-time error term $\mathcal{U}$: it is the least dynamic term (has the fewest behaviors) of every computation type, and all complex stacks are strict in errors,

Cast Universal Properties

	Bound	Best
Up	$x : A \vdash x \sqsubseteq \langle A' \nwarrow A \rangle x : A \sqsubseteq A'$	$x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \nwarrow A \rangle x \sqsubseteq x' : A'$
Down	$\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$	$\bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \langle \underline{B} \swarrow \underline{B}' \rangle \bullet : \underline{B}$

Retract Axiom

$$x : A \vdash \langle \underline{F}A \swarrow \underline{F}? \rangle(\mathtt{ret}\ (\langle ? \nwarrow A \rangle x)) \sqsubseteq \mathtt{ret}\ x : \underline{F}A$$
$$x : U\underline{B} \vdash \langle \underline{B} \swarrow \underline{¿} \rangle(\mathtt{force}\ (\langle U\underline{¿} \nwarrow U\underline{B} \rangle x)) \sqsubseteq \mathtt{force}\ x : \underline{B}$$

Type Universal Properties

Type	β	η
$+$	$\mathtt{case\ inl}\ V\{x_1.E_1 \mid \ldots\} \sqsupseteq\sqsubseteq E_1[V/x_1]$ $\mathtt{case\ inr}\ V\{\ldots \mid x_2.E_2\} \sqsupseteq\sqsubseteq E_2[V/x_2]$	$E \sqsupseteq\sqsubseteq \mathtt{case}\ x\{x_1.E[\mathtt{inl}\ x_1/x] \mid x_2.E[\mathtt{inr}\ x_2/x]\}$ where $x : A_1 + A_2 \vdash E : T$
U	$\mathtt{force\ thunk}\ M \sqsupseteq\sqsubseteq M$	$x : U\underline{B} \vdash x \sqsupseteq\sqsubseteq \mathtt{thunk\ force}\ x : U\underline{B}$
F	$\mathtt{bind}\ x \leftarrow \mathtt{ret}\ V; M \sqsupseteq\sqsubseteq M[V/x]$	$\bullet : \underline{F}A \vdash M \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow \bullet; M[\mathtt{ret}\ x/\bullet] : \underline{B}$
$\rightarrow$	$(\lambda x : A.M)\, V \sqsupseteq\sqsubseteq M[V/x]$	$\bullet : A \rightarrow \underline{B} \vdash \bullet \sqsupseteq\sqsubseteq \lambda x : A.\ \bullet\ x : A \rightarrow \underline{B}$
$\&$	$\pi\{\pi \mapsto M \mid \pi' \mapsto M'\} \sqsupseteq\sqsubseteq M$ $\pi'\{\pi \mapsto M \mid \pi' \mapsto M'\} \sqsupseteq\sqsubseteq M'$	$\bullet : \underline{B}_1 \,\&\, \underline{B}_2 \vdash \bullet \sqsupseteq\sqsubseteq \{\pi \mapsto \pi\bullet \mid \pi' \mapsto \pi'\bullet\} : \underline{B}_1 \,\&\, \underline{B}_2$

Error Properties

$$\textsc{ErrBot}\ \frac{\Gamma' \mid \cdot \vdash M' : \underline{B}'}{\Gamma \sqsubseteq \Gamma' \mid \cdot \vdash \mho \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'} \qquad \textsc{StkStrict}\ \frac{\Gamma \mid x : \underline{B} \vdash S : \underline{B}'}{\Gamma \mid \cdot \vdash S[\mho_{\underline{B}}] \sqsubseteq \mho_{\underline{B}'} : \underline{B}'}$$

Fig. 4. GTT Term Dynamism Axioms (0,×,1,⊤ in extended version)

because stacks force their evaluation position. We state the first axiom in a heterogeneous way, which includes congruence $\Gamma \sqsubseteq \Gamma' \vdash \mho_{\underline{B}} \sqsubseteq \mho_{\underline{B}'} : \underline{B} \sqsubseteq \underline{B}'$.

3 THEOREMS IN GRADUAL TYPE THEORY

In this section, we show that the axiomatics of gradual type theory determine most properties of casts, which shows that these behaviors of casts are forced in any implementation of gradual typing satisfying graduality and β, η. For proofs, see the extended version of the paper.

3.1 Type-generic Properties of Casts

The universal property axioms for upcasts and downcasts in Figure 4 define them *uniquely* up to equidynamism ($\sqsupseteq\sqsubseteq$): anything with the same property is behaviorally equivalent to a cast.

Theorem 3.1 (Specification for Casts is a Universal Property).

(1) *If* $A \sqsubseteq A'$ *and* $x : A \vdash V : A'$ *is a complex value such that* $x : A \vdash x \sqsubseteq V : A \sqsubseteq A'$ *and* $x \sqsubseteq x' : A \sqsubseteq A' \vdash V \sqsubseteq x' : A'$ *then* $x : A \vdash V \sqsupseteq\sqsubseteq \langle A' \nwarrow A \rangle x : A'$.
(2) *If* $\underline{B} \sqsubseteq \underline{B}'$ *and* $\bullet' : \underline{B}' \vdash S : \underline{B}$ *is a complex stack such that* $\bullet' : \underline{B}' \vdash S \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}'$ *and* $\bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq S : \underline{B}$ *then* $\bullet' : \underline{B}' \vdash S \sqsupseteq\sqsubseteq \langle \underline{B} \swarrow \underline{B}' \rangle \bullet' : \underline{B}$

Casts satisfy an identity and composition law:

Theorem 3.2 (Casts (de)composition). *For any* $A \sqsubseteq A' \sqsubseteq A''$ *and* $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$*:*

(1) $x : A \vdash \langle A \nwarrow A \rangle x \sqsupseteq\sqsubseteq x : A$
(2) $x : A \vdash \langle A'' \nwarrow A \rangle x \sqsupseteq\sqsubseteq \langle A'' \nwarrow A' \rangle \langle A' \nwarrow A \rangle x : A''$
(3) $\bullet : \underline{B} \vdash \langle \underline{B} \swarrow \underline{B} \rangle \bullet \sqsupseteq\sqsubseteq \bullet : \underline{B}$
(4) $\bullet : \underline{B}'' \vdash \langle \underline{B} \swarrow \underline{B}'' \rangle \bullet \sqsupseteq\sqsubseteq \langle \underline{B} \swarrow \underline{B}' \rangle (\langle \underline{B}' \swarrow \underline{B}'' \rangle \bullet) : \underline{B} \sqsubseteq \underline{B}$

In particular, this composition property implies that the casts into and out of the dynamic type are coherent, for example if $A \sqsubseteq A'$ then $\langle ? \nleftarrow A\rangle x \sqsupseteq\sqsubseteq \langle ? \nleftarrow A'\rangle\langle A' \nleftarrow A\rangle x$.

The following theorem says essentially that $x \sqsubseteq \langle T \not\leftarrow T'\rangle\langle T' \nleftarrow T\rangle x$ (upcast then downcast might error less but but otherwise does not change the behavior) and $\langle T' \nleftarrow T\rangle\langle T \not\leftarrow T'\rangle x \sqsubseteq x$ (downcast then upcast might error more but otherwise does not change the behavior). However, since a value type dynamism $A \sqsubseteq A'$ induces a value upcast $x : A \vdash \langle A' \nleftarrow A\rangle x : A'$ but a stack downcast $\bullet : \underline{F}A' \vdash \langle \underline{F}A \not\leftarrow \underline{F}A'\rangle\bullet : \underline{F}A$ (and dually for computations), the statement of the theorem wraps one cast with the constructors for U and $\underline{F}$ types (functoriality of $\underline{F}/U$).

Theorem 3.3 (Casts are a Galois Connection).

(1) $\bullet' : \underline{F}A' \vdash \mathtt{bind}\ x \leftarrow \langle \underline{F}A \not\leftarrow \underline{F}A'\rangle\bullet'; \mathtt{ret}\ (\langle A' \nleftarrow A\rangle x) \sqsubseteq \bullet' : \underline{F}A'$
(2) $\bullet : \underline{F}A \vdash \bullet \sqsubseteq \mathtt{bind}\ x \leftarrow \bullet; \langle \underline{F}A \not\leftarrow \underline{F}A'\rangle(\mathtt{ret}\ (\langle A' \nleftarrow A\rangle x)) : \underline{F}A$
(3) $x : U\underline{B}' \vdash \langle U\underline{B}' \nleftarrow U\underline{B}\rangle(\mathtt{thunk}\ (\langle \underline{B} \not\leftarrow \underline{B}'\rangle \mathtt{force}\ x)) \sqsubseteq x : U\underline{B}'$
(4) $x : U\underline{B} \vdash x \sqsubseteq \mathtt{thunk}\ (\langle B \not\leftarrow B'\rangle(\mathtt{force}\ (\langle U\underline{B}' \nleftarrow U\underline{B}\rangle x))) : U\underline{B}$

The retract property says roughly that $x \sqsupseteq\sqsubseteq \langle T' \not\leftarrow T\rangle\langle T' \nleftarrow T\rangle x$ (upcast then downcast does not change the behavior), strengthening the $\sqsubseteq$ of Theorem 3.3. In Figure 4, we asserted the retract axiom for casts with the dynamic type. This and the composition property implies the retraction property for general casts:

Theorem 3.4 (Retract Property for General Casts).

(1) $\bullet : \underline{F}A \vdash \mathtt{bind}\ x \leftarrow \bullet; \langle \underline{F}A \not\leftarrow \underline{F}A'\rangle(\mathtt{ret}\ (\langle A' \nleftarrow A\rangle x)) \sqsupseteq\sqsubseteq \bullet : \underline{F}A$
(2) $x : U\underline{B} \vdash \mathtt{thunk}\ (\langle \underline{B} \not\leftarrow \underline{B}'\rangle(\mathtt{force}\ (\langle U\underline{B}' \nleftarrow U\underline{B}\rangle x))) \sqsupseteq\sqsubseteq x : U\underline{B}$

3.2 Unique Implementations of Casts

Together, the universal property for casts and the η principles for each type imply that the casts must behave as in lazy cast semantics:

Theorem 3.5 (Cast Unique Implementation Theorem for $+, \times, \to, \&$). *The casts' behavior is uniquely determined as follows: (See the extended version for $+$, $\&$.)*

$$\langle A_1' \times A_2' \nleftarrow A_1 \times A_2\rangle p \sqsupseteq\sqsubseteq \mathtt{split}\ p\ \mathtt{to}\ (x_1, x_2).(\langle A_1' \nleftarrow A_1\rangle x_1, \langle A_2' \nleftarrow A_2\rangle x_2)$$

$$\langle \underline{F}(A_1' \times A_2') \not\leftarrow \underline{F}(A_1 \times A_2)\rangle\bullet \quad \sqsupseteq\sqsubseteq \quad \begin{array}{l}\mathtt{bind}\ p' \leftarrow \bullet; \mathtt{split}\ p'\ \mathtt{to}\ (x_1', x_2').\\ \mathtt{bind}\ x_1 \leftarrow \langle \underline{F}A_1 \not\leftarrow \underline{F}A_1'\rangle\mathtt{ret}\ x_1';\\ \mathtt{bind}\ x_2 \leftarrow \langle \underline{F}A_2 \not\leftarrow \underline{F}A_2'\rangle\mathtt{ret}\ x_2'; \mathtt{ret}\ (x_1, x_2)\end{array}$$

$$\langle A \to \underline{B} \not\leftarrow A' \to \underline{B}'\rangle\bullet \sqsupseteq\sqsubseteq \lambda x.\langle \underline{B} \not\leftarrow \underline{B}'\rangle(\bullet\,(\langle A' \nleftarrow A\rangle x))$$

$$\langle U(A' \to \underline{B}') \nleftarrow U(A \to \underline{B})\rangle f \quad \sqsupseteq\sqsubseteq \quad \begin{array}{ll}\mathtt{thunk}\ (\lambda x'. & \mathtt{bind}\ x \leftarrow \langle \underline{F}A \not\leftarrow \underline{F}A'\rangle(\mathtt{ret}\ x');\\ & \mathtt{force}\ (\langle U\underline{B}' \nleftarrow U\underline{B}\rangle(\mathtt{thunk}\ (\mathtt{force}\ (f)\, x))))\end{array}$$

In the case for an eager product $\times$, we can actually also show that reversing the order and running $\langle \underline{F}A_2 \not\leftarrow \underline{F}A_2'\rangle\mathtt{ret}\ x_2'$ and then $\langle \underline{F}A_1 \not\leftarrow \underline{F}A_1'\rangle\mathtt{ret}\ x_1'$ is also an implementation of this cast, and therefore equal to the above. Intuitively, this is sensible because the only effect a downcast introduces is a run-time error, and if either downcast errors, both possible implementations will.

In GTT, we assert the existence of value upcasts and computation downcasts for derivable type dynamism relations. While we do not assert the existence of all *value* downcasts and *computation* upcasts, we can define the universal property that identifies a term as such:

Definition 3.6 (Stack upcasts/value downcasts).

(1) If $\underline{B} \sqsubseteq \underline{B}'$, a *stack upcast from B to B'* is a stack $\bullet : \underline{B} \vdash \langle \underline{B}' \nwarrow \underline{B} \rangle \bullet : \underline{B}'$ that satisfies the computation dynamism rules of an upcast $\bullet : \underline{B} \vdash \bullet \sqsubseteq \langle\!\langle \underline{B}' \nwarrow \underline{B} \rangle\!\rangle \bullet : \underline{B} \sqsubseteq \underline{B}'$ and $\bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \vdash \langle\!\langle \underline{B}' \nwarrow \underline{B} \rangle\!\rangle \bullet \sqsubseteq \bullet' : \underline{B}'$.
(2) If $A \sqsubseteq A'$, a *value downcast from A' to A* is a complex value $x : A' \vdash \langle\!\langle A \swarrow A' \rangle\!\rangle x : A$ that satisfies the value dynamism rules of a downcast $x : A' \vdash \langle\!\langle A \swarrow A' \rangle\!\rangle x \sqsubseteq x : A \sqsubseteq A'$ and $x \sqsubseteq x' : A \sqsubseteq A' \vdash x \sqsubseteq \langle\!\langle A \swarrow A' \rangle\!\rangle x' : A$.

Some value downcasts and computation upcasts do exist, leading to a characterization of the casts for the monad $U\underline{F}A$ and comonad $\underline{F}U\underline{B}$ of $F \dashv U$:

Theorem 3.7 (Cast Unique Implementation Theorem for $U\underline{F}, \underline{F}U$). *Let $A \sqsubseteq A'$ and $\underline{B} \sqsubseteq \underline{B}'$.*

(1) $\bullet : \underline{F}A \vdash \mathtt{bind}\ x : A \leftarrow \bullet; \mathtt{ret}\ (\langle A' \nwarrow A \rangle x) : \underline{F}A'$ *is a stack upcast.*
(2) *If $\langle\!\langle \underline{B}' \nwarrow \underline{B} \rangle\!\rangle$ is a stack upcast, then*
$x : U\underline{B} \vdash \langle U\underline{B}' \nwarrow U\underline{B} \rangle x \sqsupseteq \sqsubseteq \mathtt{thunk}\ (\langle\!\langle \underline{B}' \nwarrow \underline{B} \rangle\!\rangle (\mathtt{force}\ x)) : U\underline{B}'$
(3) $x : U\underline{B}' \vdash \mathtt{thunk}\ (\langle \underline{B} \swarrow \underline{B}' \rangle (\mathtt{force}\ x)) : U\underline{B}$ *is a value downcast.*
(4) *If $\langle\!\langle A \swarrow A' \rangle\!\rangle$ is a value downcast, then*
$\bullet : \underline{F}A' \vdash \langle \underline{F}A \swarrow \underline{F}A' \rangle \bullet \sqsupseteq \sqsubseteq \mathtt{bind}\ x' : A' \leftarrow \bullet; \mathtt{ret}\ (\langle A \swarrow A' \rangle x)$
(5) $x : U\underline{F}A \vdash \langle U\underline{F}A' \nwarrow U\underline{F}A \rangle x \sqsupseteq \sqsubseteq \mathtt{thunk}\ (\mathtt{bind}\ x : A \leftarrow \mathtt{force}\ x; \mathtt{ret}\ (\langle A' \nwarrow A \rangle x))$
$\bullet : \underline{F}U\underline{B}' \vdash \langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}' \rangle \bullet \sqsupseteq \sqsubseteq \mathtt{bind}\ x' : U\underline{B}' \leftarrow \bullet; \mathtt{ret}\ (\mathtt{thunk}\ (\langle \underline{B} \swarrow \underline{B}' \rangle (\mathtt{force}\ x)))$

Recall that for value types A_1 and A_2, the CBV function type is $U(A_1 \to \underline{F}A_2)$. As a corollary of Theorems 3.5 and 3.7, we have

Corollary 3.8 (Cast Unique Implementation for CBV Functions).

$$\begin{array}{lll} \langle U(A_1' \to \underline{F}A_2') \nwarrow U(A_1 \to \underline{F}A_2) \rangle f & \sqsupseteq \sqsubseteq & \mathtt{thunk}\ (\lambda x'.\ \ \mathtt{bind}\ x \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle (\mathtt{ret}\ x'); \\ & & \quad\ \mathtt{bind}\ y \leftarrow (\mathtt{force}\ (f)\, x); \\ & & \quad\ \mathtt{ret}\ (\langle A_2' \nwarrow A_2 \rangle y)) \\ \langle \underline{F}U(A_1 \to \underline{F}A_2) \swarrow \underline{F}U(A_1' \to \underline{F}A_2') \rangle \bullet & \sqsupseteq \sqsubseteq & \mathtt{bind}\ f \leftarrow \bullet; \\ & & \mathtt{ret}\ \lambda x. \langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle (\mathtt{force}\ (f)\, (\langle A_1' \nwarrow A_1 \rangle x)) \end{array}$$

These are equivalent to the CBPV translations of the standard CBV wrapping implementations; for example, the CBV upcast term $\lambda x'.\mathtt{let}\ x = \langle A_1 \swarrow A_1' \rangle x'; \langle A_2' \nwarrow A_2 \rangle (f x')$ has its evaluation order made explicit, and the fact that its upcast is a (complex) value exposed. In the downcast, the GTT term is free to let-bind $(\langle A_1' \nwarrow A_1 \rangle x)$ to avoid duplicating it, but because it is a (complex) value, it can also be substituted directly, which might expose reductions that can be optimized.

3.3 Upcasts are Values, Downcasts are Stacks

Since GTT is an axiomatic theory, we can consider different fragments than the one presented in Section 2. Here, we use this flexibility to show that taking upcasts to be complex values and downcasts to be complex stacks is forced if this property holds for casts between *ground* types and $?/\underline{¿}$. For this section, we define a *ground type*[3] to be generated by the following grammar:

$$G ::= 1 \mid ? \times ? \mid 0 \mid ? + ? \mid U\underline{¿} \qquad \underline{G} ::= ? \to \underline{¿} \mid \top \mid \underline{¿}\ \&\ \underline{¿} \mid \underline{F}?$$

Let GTT_G be the fragment of GTT where the only primitive casts are those between ground types and the dynamic types, i.e. the cast terms are restricted to the substitution closures of

$$x : G \vdash \langle ? \nwarrow G \rangle x : ? \qquad \bullet : \underline{F}? \vdash \langle \underline{F}G \swarrow \underline{F}? \rangle \bullet : \underline{F}? \qquad \bullet : \underline{¿} \vdash \langle \underline{G} \swarrow \underline{¿} \rangle \bullet : \underline{¿} \qquad x : U\underline{G} \vdash \langle U\underline{¿} \nwarrow U\underline{G} \rangle x : U\underline{¿}$$

Lemma 3.9 (Casts are Admissible). *In GTT_G it is admissible that*

[3] In gradual typing, "ground" is used to mean a one-level unrolling of a dynamic type, not first-order data.

(1) *for all* $A \sqsubseteq A'$ *there is a complex value* $\langle\!\langle A' \nwarrow A \rangle\!\rangle$ *satisfying the universal property of an upcast and a complex stack* $\langle\!\langle \underline{F}A \swarrow \underline{F}A' \rangle\!\rangle$ *satisfying the universal property of a downcast*
(2) *for all* $\underline{B} \sqsubseteq \underline{B}'$ *there is a complex stack* $\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle$ *satisfying the universal property of a downcast and a complex value* $\langle\!\langle U\underline{B}' \nwarrow U\underline{B} \rangle\!\rangle$ *satisfying the universal property of an upcast.*

Proof. To streamline the exposition above, we stated Theorems 3.2, Theorem 3.5 Theorem 3.7 as showing that the "definitions" of each cast are equidynamic with the cast that is a priori postulated to exist (e.g. $\langle A'' \nwarrow A \rangle \sqsupseteq\sqsubseteq \langle A'' \nwarrow A' \rangle \langle A' \nwarrow A \rangle$). However, the proofs show directly that the right-hand sides have the desired universal property—i.e. the stipulation that some cast with the correct universal property exists is not used in the proof that the implementation has the desired universal property. Moreover, the proofs given do not rely on any axioms of GTT besides the universal properties of the "smaller" casts used in the definition and the $\beta\eta$ rules for the relevant types. So these proofs can be used as the inductive steps here, in GTT_G. In the extended version we define an alternative type dynamism relation where casts into dynamic types are factored through ground types, and use that to drive the induction here. □

As discussed in Section 2.4, rather than an upcast being a complex value $x : A \vdash \langle A' \nwarrow A \rangle x : A'$, an a priori more general type would be a stack $\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet : \underline{F}A'$, which allows the upcast to perform effects; dually, an a priori more general type for a downcast $\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet : \underline{B}$ would be a value $x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x : U\underline{B}$, which allows the downcast to ignore its argument. The following shows that in GTT_G, if we postulate such stack upcasts/value downcasts as originally suggested in Section 2.4, then in fact these casts *must* be equal to the action of $U/\underline{F}$ on some value upcasts/stack downcasts, so the potential for (co)effectfulness affords no additional flexibility.

Theorem 3.10 (Upcasts are Necessarily Values, Downcasts are Necessarily Stacks). *Suppose we extend* GTT_G *with the following postulated stack upcasts and value downcasts (in the sense of Definition 3.6): For every type precision* $A \sqsubseteq A'$*, there is a stack upcast* $\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet : \underline{F}A'$*, and for every* $\underline{B} \sqsubseteq \underline{B}'$*, there is a complex value downcast* $x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x : U\underline{B}$*.*

Then there exists a value upcast $\langle\!\langle A' \nwarrow A \rangle\!\rangle$ *and a stack downcast* $\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle$ *such that*

$$\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet \sqsupseteq\sqsubseteq (\mathtt{bind}\ x : A \leftarrow \bullet; \mathtt{ret}\ (\langle\!\langle A' \nwarrow A \rangle\!\rangle x))$$
$$x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x \sqsupseteq\sqsubseteq (\mathtt{thunk}\ (\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle (\mathtt{force}\ x)))$$

Proof. Lemma 3.9 constructs $\langle\!\langle A' \nwarrow A \rangle\!\rangle$ and $\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle$, so the proof of Theorem 3.7 (which really works for any $\langle\!\langle A' \nwarrow A \rangle\!\rangle$ and $\langle\!\langle \underline{B} \swarrow \underline{B}' \rangle\!\rangle$ with the correct universal properties, not only the postulated casts) implies that the right-hand sides of the above equations are stack upcasts and value downcasts of the appropriate type. Since stack upcasts/value downcasts are unique by an argument analogous to Theorem 3.1, the postulated casts must be equal to these. □

4 CONTRACT MODELS OF GTT

To show the soundness of our theory, and demonstrate its relationship to operational definitions of observational equivalence and the gradual guarantee, we develop *models* of GTT using observational error approximation of a *non-gradual* CBPV. We call this the *contract translation* because it translates the built-in casts of the gradual language into ordinary terms implemented in a non-gradual language. While contracts are typically implemented in a dynamically typed language, our target is typed, retaining type information similarly to manifest contracts [Greenberg et al. 2010]. We give implementations of the dynamic value type in the usual way as a recursive sum of basic value types, i.e., using type tags, and we give implementations of the dynamic computation type as the dual: a recursive product of basic computation types.

Writing $[\![M]\!]$ for any of the contract translations, the remaining sections of the paper establish:

Theorem 4.1 (Equi-dynamism implies Observational Equivalence). *If* $\Gamma \vdash M_1 \sqsupseteq\sqsubseteq M_2 : \underline{B}$, *then for any closing GTT context* $C : (\Gamma \vdash \underline{B}) \Rightarrow (\cdot \vdash \underline{F}(1+1))$, $[\![C[M_1]]\!]$ *and* $[\![C[M_2]]\!]$ *have the same behavior: both diverge, both run to an error, or both run to* `true` *or both run to* `false`.

Theorem 4.2 (Graduality). *If* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash M_1 \sqsubseteq M_2 : B_1 \sqsubseteq B_2$, *then for any GTT context* $C : (\Gamma_1 \vdash B_1) \Rightarrow (\cdot \vdash \underline{F}(1+1))$, *and any valid interpretation of the dynamic types, either*

(1) $[\![C[M_1]]\!] \Downarrow \mho$, *or*
(2) $[\![C[M_1]]\!] \Uparrow$ *and* $[\![C[\langle B_1 \swarrow B_2 \rangle M_2[\langle \Gamma_2 \nwarrow \Gamma_1 \rangle \Gamma_1]]]\!] \Uparrow$, *or*
(3) $[\![C[M_1]]\!] \Downarrow \texttt{ret}\ V$, $[\![C[\langle B_1 \swarrow B_2 \rangle M_2[\langle \Gamma_2 \nwarrow \Gamma_1 \rangle \Gamma_1]]]\!] \Downarrow \texttt{ret}\ V$, *and* $V = \texttt{true}$ *or* $V = \texttt{false}$.

As a consequence we will also get consistency of our logic of dynamism:

Corollary 4.3 (Consistency). $\cdot \vdash \texttt{ret true} \sqsubseteq \texttt{ret false} : \underline{F}(1+1)$ *is not provable in GTT.*

We break down this proof into 3 major steps.

(1) (This section) We translate GTT into a statically typed CBPV* language where the casts of GTT are translated to "contracts" in GTT: i.e., CBPV terms that implement the runtime type checking. We translate the term dynamism of GTT to an inequational theory for CBPV. Our translation is parameterized by the implementation of the dynamic types, and we demonstrate two valid implementations, one more direct and one more Scheme-like.
(2) (Section 5) Next, we eliminate all uses of complex values and stacks from the CBPV language. We translate the complex values and stacks to terms with a proof that they are "pure" (thunkable or linear [Munch-Maccagnoni 2014]). This part has little to do with GTT specifically, except that it shows the behavioral property that corresponds to upcasts being complex values and downcasts being complex stacks.
(3) (Section 6.3) Finally, with complex values and stacks eliminated, we give a standard operational semantics for CBPV and define a *logical relation* that is sound and complete with respect to observational error approximation. Using the logical relation, we show that the inequational theory of CBPV is sound for observational error approximation.

By composing these, we get a model of GTT where equidynamism is sound for observational equivalence and an operational semantics that satisfies the graduality theorem.

4.1 Call-by-push-value

Next, we define the call-by-push-value language CBPV* that will be the target for our contract translations of GTT. CBPV* is the axiomatic version of call-by-push-value *with* complex values and stacks, while CBPV (Section 5) will designate the operational version of call-by-push-value with only operational values and stacks. CBPV* is almost a subset of GTT obtained as follows: We remove the casts and the dynamic types ?, ¿ (the shaded pieces) from the syntax and typing rules in Figure 1. There is no type dynamism, and the inequational theory of CBPV* is the homogeneous fragment of term dynamism in Figure 3 (judgements $\Gamma \vdash E \sqsubseteq E' : T$ where $\Gamma \vdash E, E' : T$, with all the same rules in that figure thus restricted). The inequational axioms are the Type Universal Properties ($\beta\eta$ rules) and Error Properties (with ErrBot made homogeneous) from Figure 4. To implement the casts and dynamic types, we *add* general *recursive* value types ($\mu X.A$, the fixed point of X val type $\vdash A$ val type) and *corecursive* computation types ($\nu \underline{Y}.\underline{B}$, the fixed point of $\underline{Y}$ comp type $\vdash \underline{B}$ comp type). The recursive type $\mu X.A$ is a value type with constructor `roll`, whose eliminator is pattern matching, whereas the corecursive type $\nu \underline{Y}.\underline{B}$ is a computation type defined by its eliminator (`unroll`), with an introduction form that we also write as `roll`. We extend the inequational theory with monotonicity of each term constructor of the recursive types, and with their $\beta\eta$ rules. The rules for recursive types are in the extended version.

4.2 Interpreting the Dynamic Types

As shown in Theorems 3.2, 3.5, 3.7, almost all of the contract translation is uniquely determined already. However, the interpretation of the dynamic types and the casts between the dynamic types and ground types G and $\underline{G}$ are not determined (they were still postulated in Lemma 3.9). For this reason, our translation is *parameterized* by an interpretation of the dynamic types and the ground casts. By Theorems 3.3, 3.4, we know that these must be *embedding-projection pairs* (ep pairs), which we now define in CBPV*.

Definition 4.4 (Value and Computation Embedding-Projection Pairs).

(1) A *value ep pair* from A to A' consists of an *embedding* value $x : A \vdash V_e : A'$ and *projection* stack $\bullet : \underline{F}A' \vdash S_p : \underline{F}A$, satisfying the *retraction* and *projection* properties:

$$x : A \vdash \mathtt{ret}\ x \sqsupseteq\sqsubseteq S_p[\mathtt{ret}\ V_e] : \underline{F}A \qquad \bullet : \underline{F}A' \vdash \mathtt{bind}\ x \leftarrow S_p; \mathtt{ret}\ V_e \sqsubseteq \bullet : \underline{F}A'$$

(2) A *computation ep pair* from $\underline{B}$ to $\underline{B}'$ consists of an *embedding* value $z : U\underline{B} \vdash V_e : U\underline{B}'$ and a *projection* stack $\bullet : \underline{B}' \vdash S_p : \underline{B}$ satisfying *retraction* and *projection* properties:

$$z : U\underline{B} \vdash \mathtt{force}\ z \sqsupseteq\sqsubseteq S_p[\mathtt{force}\ V_e] : \underline{B} \qquad w : U\underline{B}' \vdash V_e[\mathtt{thunk}\ S_p[\mathtt{force}\ w]] \sqsubseteq w : U\underline{B}'$$

Using this, and using the notion of ground type from Section 3.3 *with* 0 *and* $\top$ *removed*, we define

Definition 4.5 (Dynamic Type Interpretation). A $?, \underline{¿}$ interpretation ρ consists of (1) a CBPV value type $\rho(?)$, (2) a CBPV computation type $\rho(\underline{¿})$, (3) for each value ground type G, a value ep pair $(x.\rho_e(G), \rho_p(G))$ from $[\![G]\!]_\rho$ to $\rho(?)$, and (4) for each computation ground type $\underline{G}$, a computation ep pair $(z.\rho_e(\underline{G}), \rho_p(\underline{G}))$ from $[\![\underline{G}]\!]_\rho$ to $\rho(\underline{¿})$. We write $[\![G]\!]_\rho$ and $[\![\underline{G}]\!]_\rho$ for the interpretation of a ground type, replacing $?$ with $\rho(?)$, $\underline{¿}$ with $\rho(\underline{¿})$, and compositionally otherwise.

Next, we show several possible interpretations of the dynamic type that will all give, by construction, implementations that satisfy the gradual guarantee. Our interpretations of the value dynamic type are not surprising. They are the usual construction of the dynamic type using type tags: i.e., a recursive sum of basic value types. On the other hand, our interpretations of the computation dynamic type are less familiar. In duality with the interpretation of $?$, we interpret $\underline{¿}$ as a recursive *product* of basic computation types. This interpretation has some analogues in previous work on the duality of computation [Girard 2001; Zeilberger 2009], but the most direct interpretation (definition 4.8) does not correspond to any known work on dynamic/gradual typing. Then we show that a particular choice of which computation types is basic and which are derived produces an interpretation of the dynamic computation type as a type of variable-arity functions whose arguments are passed on the stack, producing a model similar to Scheme without accounting for control effects (definition 4.9).

4.2.1 Natural Dynamic Type Interpretation. Our first dynamic type interpretation is to make the value and computation dynamic types sums and products of the ground value and computation types, respectively. This forms a model of GTT for the following reasons. For the value dynamic type $?$, we need a value embedding (the upcast) from each ground value type G with a corresponding projection. The easiest way to do this would be if for each G, we could rewrite $?$ as a sum of the values that fit G and those that don't: $? \cong G + ?_{-G}$ because of the following lemma.

Lemma 4.6 (Sum Injections are Value Embeddings). *For any A, A', there are value ep pairs from A and A' to $A + A'$ where the embeddings are* $\mathtt{inl}$ *and* $\mathtt{inr}$ *.*

Proof. Define the embedding of A to just be $x.\mathtt{inl}\ x$ and the projection to be $\mathtt{bind}\ y \leftarrow \bullet; \mathtt{case}\ y\{\mathtt{inl}\ x.\mathtt{ret}\ x \mid \mathtt{inr}\ \_.\mho\}$. □

This shows why the type tag interpretation works: it makes the dynamic type in some sense the minimal type with injections from each G: the sum of all value ground types $? \cong \Sigma_G G$.

The dynamic computation type $\underline{¿}$ can be naturally defined by a dual construction, by the following dual argument. First, we want a computation ep pair from $\underline{G}$ to $\underline{¿}$ for each ground computation type $\underline{G}$. Specifically, this means we want a stack from $\underline{¿}$ to $\underline{G}$ (the downcast) with an embedding. The easiest way to get this is if, for each ground computation type $\underline{G}$, $\underline{¿}$ is equivalent to a lazy product of $\underline{G}$ and "the other behaviors", i.e., $\underline{¿} \cong \underline{G} \mathbin{\&} \underline{¿}_{-\underline{G}}$. Then the embedding on π performs the embedded computation, but on π' raises a type error. The following lemma, dual to lemma 4.6 shows this forms a computation ep pair:

Lemma 4.7 (Lazy Product Projections are Computation Projections). *For any $\underline{B}, \underline{B}'$, there are computation ep pairs from $\underline{B}$ and $\underline{B}'$ to $\underline{B} \mathbin{\&} \underline{B}'$ where the projections are π and π'.*

Proof. Define the projection for $\underline{B}$ to be π. Define the embedding by $z.\{\pi \mapsto \texttt{force}\ z \mid \pi' \mapsto \mathcal{U}\}$. Similarly define the projection for $\underline{B}'$. □

From this, we see that the easiest way to construct an interpretation of the dynamic computation type is to make it a lazy product of all the ground types $\underline{G}$: $\underline{¿} \cong \&_{\underline{G}}\, \underline{G}$. Using recursive types, we can easily make this a definition of the interpretations:

Definition 4.8 (Natural Dynamic Type Interpretation). The following defines a dynamic type interpretation. We define the types to satisfy the isomorphisms

$$? \cong 1 + (? \times ?) + (? + ?) + U\underline{¿} \qquad \underline{¿} \cong (\underline{¿} \mathbin{\&} \underline{¿}) \mathbin{\&} (? \to \underline{¿}) \mathbin{\&} \underline{F}?$$

with the ep pairs defined as in Lemma 4.6 and 4.7.

This dynamic type interpretation is a natural fit for CBPV because the introduction forms for ? are exactly the introduction forms for all of the value types (unit, pairing,`inl`, `inr`, `force`), while elimination forms are all of the elimination forms for computation types (π, π', application and binding); such "bityped" languages are related to Girard [2001]; Zeilberger [2009]. In the extended version, we give an extension of GTT axiomatizing this implementation of the dynamic types.

4.2.2 Scheme-like Dynamic Type Interpretation. The above dynamic type interpretation does not correspond to any dynamically typed language used in practice, in part because it includes explicit cases for the "additives", the sum type + and lazy product type &. Normally, these are not included in this way, but rather sums are encoded by making each case use a fresh constructor (using nominal techniques like opaque structs in Racket) and then making the sum the union of the constructors, as argued in Siek and Tobin-Hochstadt [2016]. We leave modeling this nominal structure to future work, but in minimalist languages, such as simple dialects of Scheme and Lisp, sum types are often encoded *structurally* rather than nominally by using some fixed sum type of *symbols*, also called *atoms*. Then a value of a sum type is modeled by a pair of a symbol (to indicate the case) and a payload with the actual value. We can model this by using the canonical isomorphisms

$$? + ? \cong ((1 + 1) \times ?) \qquad \underline{¿} \mathbin{\&} \underline{¿} \cong (1 + 1) \to \underline{¿}$$

and representing sums as pairs, and lazy products as functions.

With this in mind, we remove the cases for sums and lazy pairs from the natural dynamic types, and include some atomic type as a case of ?—for simplicity we will just use booleans. We also do not need a case for 1, because we can identify it with one of the booleans, say `true`. This leads to the following definition:

$$1 \sqsubseteq \mathbb{B} \qquad A + A \sqsupseteq\sqsubseteq \mathbb{B} \times A \qquad \underline{B} \,\&\, \underline{B} \sqsupseteq\sqsubseteq \mathbb{B} \to \underline{B}$$

$$\frac{\Gamma \mid \Delta \vdash M_{\to} : ? \to \underline{¿} \qquad \Gamma \mid \Delta \vdash M_{\underline{F}} : \underline{F}?}{\Gamma \mid \Delta \vdash \{(\to) \mapsto M_{\to} \mid \underline{F} \mapsto M_{\underline{F}}\} : \underline{¿}}\ \underline{¿}\mathrm{I}$$

$$\frac{\Gamma \mid \Delta \vdash V : ? \qquad \Gamma, x_{\mathbb{B}} : \mathbb{B} \mid \Delta \vdash E_{\mathbb{B}} : T \qquad \Gamma, x_U : U\underline{¿} \mid \Delta \vdash E_U : T \qquad \Gamma, x_\times : ? \times ? \mid \Delta \vdash E_\times : T}{\Gamma \mid \Delta \vdash \mathsf{tycase}\ V\ \{x_{\mathbb{B}}.E_{\mathbb{B}} \mid x_U.E_U \mid x_\times.E_\times\} : T}\ ?\mathrm{E}$$

Fig. 5. Scheme-like Extension to GTT

Definition 4.9 (Scheme-like Dynamic Type Interpretation). We can define a dynamic type interpretation with the following type isomorphisms:

$$? \cong (1 + 1) + U\underline{¿} + (? \times ?) \qquad\qquad \underline{¿} \cong (? \to \underline{¿}) \,\&\, \underline{F}?$$

Proof. The details of constructing the two mutually recursive types from our recursive type mechanism are in the extended version. The ep pairs for $\times, U, \underline{F}, \to$ are clear. To define the rest, first note that there is an ep pair from $1 + 1$ to $?$ by Lemma 4.6. Next, we can define 1 to be the ep pair to $1 + 1$ defined by the left case and Lemma 4.6, composed with this. The ep pair for $? + ?$ is defined by composing the isomorphism (which is always an ep pair) $(? + ?) \cong ((1 + 1) \times ?)$ with the ep pair for $1 + 1$ using the action of product types on ep pairs (proven as part of Theorem 4.11): $(? + ?) \cong ((1 + 1) \times ?) \triangleleft (? \times ?) \triangleleft ?$ (where we write $A \triangleleft A'$ to mean there is an ep pair from A to A'). Similarly, for $\underline{¿} \,\&\, \underline{¿}$, we use action of the function type on ep pairs (also proven as part of Theorem 4.11): $\underline{¿} \,\&\, \underline{¿} \cong ((1 + 1) \to \underline{¿}) \triangleleft (? \to \underline{¿}) \triangleleft \underline{¿}$ □

Intuitively, the above definition of ? says that it is a binary tree whose leaves are either booleans or closures—a simple type of S-expressions. On the other hand, the above definition of $\underline{¿}$ models a *variable-arity function* (as in Scheme), which is called with any number of dynamically typed value arguments ? and returns a dynamically typed result $\underline{F}?$. To see why a $\underline{¿}$ can be called with any number of arguments, observe that its infinite unrolling is $\underline{F}? \,\&\, (? \to \underline{F}?) \,\&\, (? \to ? \to \underline{F}?) \,\&\, \ldots$. This type is isomorphic to a function that takes a list of ? as input $((\mu X.1 + (? \times X)) \to \underline{F}?)$, but operationally $\underline{¿}$ is a more faithful model of Scheme implementations, because all of the arguments are passed individually on the stack, not as a heap-allocated single list argument. These two are distinguished in Scheme and the "dot args" notation witnesses the isomorphism.

Based on this dynamic type interpretation we can make a "Scheme-like" extension to GTT in Figure 5. First, we add a boolean type $\mathbb{B}$ with `true`, `false` and if-then-else. Next, we add in the elimination form for ? and the introduction form for $\underline{¿}$. The elimination form for ? is a typed version of Scheme's *match* macro. The introduction form for $\underline{¿}$ is a typed, CBPV version of Scheme's *case-lambda* construct. Finally, we add type dynamism rules expressing the representations of 1, $A + A$, and $A \times A$ in terms of booleans that were explicit in the ep pairs used in Definition 4.9. In the extended version of the paper, we include the appropriate term dynamism axioms, which are straightforward syntactifications of the properties of the dynamic type interpretation, and prove a unique implementation theorem for the new casts.

4.3 Contract Translation

Having defined the data parameterizing the translation, we now consider the translation of GTT into CBPV* itself. For the remainder of the paper, we assume that we have a fixed dynamic type interpretation ρ, and all proofs and definitions work for any interpretation.

The main idea of the translation is an extension of the dynamic type interpretation to an interpretation of *all* casts in GTT (Figure 6) as contracts in CBPV*, following the definitions in

$$x : [\![A]\!] \vdash [\![\langle A' \nwarrow A\rangle]\!] : [\![A']\!] \qquad \bullet : [\![\underline{B}']\!] \vdash [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!] : [\![\underline{B}]\!]$$

$$\begin{array}{rcl}
x : [\![?]\!] \vdash [\![\langle ? \nwarrow ?\rangle]\!] & = & x \\
\bullet : \underline{F}? \vdash [\![\langle \underline{F}? \swarrow \underline{F}?\rangle]\!] & = & \bullet \\
x : [\![G]\!] \vdash [\![\langle ? \nwarrow G\rangle]\!] & = & \rho_{up}(G) \\
\bullet : \underline{F}? \vdash [\![\langle \underline{F}G \swarrow \underline{F}?\rangle]\!] & = & \rho_{dn}(G) \\
x : [\![A]\!] \vdash [\![\langle ? \nwarrow A\rangle]\!] & = & [\![\langle ? \nwarrow \lfloor A\rfloor\rangle]\!][[\![\langle \lfloor A\rfloor \nwarrow A\rangle]\!]/x] \\
\bullet : \underline{F}? \vdash [\![\langle A \swarrow ?\rangle]\!] & = & [\![\langle A \swarrow \lfloor A\rfloor\rangle]\!][[\![\langle \lfloor A\rfloor \swarrow ?\rangle]\!]] \\
x : [\![A_1]\!] \times [\![A_2]\!] \vdash [\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle]\!] & = & \texttt{split } x \texttt{ to } (x_1, x_2). \\
 & & ([\![\langle A_1' \nwarrow A_1\rangle]\!][x_1], [\![\langle A_2' \nwarrow A_2\rangle]\!][x_2]) \\
\bullet \vdash [\![\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle]\!] & = & \texttt{bind } x' \leftarrow \bullet; \texttt{split } x' \texttt{ to } (x_1', x_2'). \\
 & & \texttt{bind } x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle]\!]\texttt{ret } x_1'; \\
 & & \texttt{bind } x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle]\!]\texttt{ret } x_2'; \texttt{ret } (x_1, x_2) \\
x : U\underline{F}[\![A]\!] \vdash [\![\langle U\underline{F}A' \nwarrow U\underline{F}A\rangle]\!] & = & \texttt{thunk } (\texttt{bind } y \leftarrow \texttt{force } x; \texttt{ret } [\![\langle A' \nwarrow A\rangle]\!][y/x])
\end{array}$$

Fig. 6. Cast to Contract Translation (selected cases)

Lemma 3.9. To verify the totality and coherence of this definition, we define (in the extended version) a normalized version of the type dynamism rules from Figure 2, which is interderivable but has at most one derivation of $T \sqsubseteq T'$ for a given T and T'. The main idea is to restrict reflexivity to base types, and restrict transitivity to $A \sqsubseteq \lfloor A\rfloor \sqsubseteq ?$, where $\lfloor A\rfloor$ is the ground type with the same outer connective as A. Next, we extend the translation of casts to a translation of all terms by congruence, since all terms in GTT besides casts are in CBPV*. This satisfies:

Lemma 4.10 (Contract Translation Type Preservation). *If $\Gamma \mid \Delta \vdash E : T$ in GTT, then $[\![\Gamma]\!] \mid [\![\Delta]\!] \vdash [\![E]\!] : [\![T]\!]$ in CBPV*.*

We have now given an interpretation of the types, terms, and type dynamism proofs of GTT in CBPV*. To complete this to form a *model* of GTT, we need to give an interpretation of the *term dynamism* proofs, which is established by the following "axiomatic graduality" theorem. GTT has *heterogeneous* term dynamism rules indexed by type dynamism, but CBPV* has only *homogeneous* inequalities between terms, i.e., if $E \sqsubseteq E'$, then E, E' have the *same* context and types. Since every type dynamism judgement has an associated contract, we can translate a heterogeneous term dynamism to a homogeneous inequality *up to contract*. Our next overall goal is to prove

Theorem 4.11 (Axiomatic Graduality). *For any dynamic type interpretation,*

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Psi : \Delta \sqsubseteq \Delta' \qquad \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{[\![\Gamma]\!] \mid [\![\Delta']\!] \vdash [\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]]] : [\![\underline{B}]\!]} \qquad \frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{[\![\Gamma]\!] \vdash [\![\langle A' \nwarrow A\rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]] : [\![A']\!]}$$

where we define $[\![\Phi]\!]$ to upcast each variable, and $[\![\Delta]\!]$ to downcast $\bullet$ if it is nonempty, and if $\Delta = \cdot$, then $M[[\![\Delta]\!]] = M$.

The full proof can be found in the extended version, and uses a sequence of lemmas. The first lemma shows that the translations of casts in Figure 6 do form ep pairs in the sense of Definition 4.4. One of the biggest advantages of using an explicit syntax for complex values and complex stacks is that the "shifted" casts (the downcast between $\underline{F}$ types for $A \sqsubseteq A'$ and the upcast between U types for $\underline{B} \sqsubseteq \underline{B}'$) are the only effectful terms, and this lemma is the only place where we need to reason about their definitions explicitly—afterwards, we can simply use the fact that they are ep pairs with the "pure" value upcasts and stack downcasts, which compose much more nicely than effectful terms. This is justified by two additional lemmas, which show that a projection is determined by its embedding and vice versa, and that embedding-projections satisfy an adjunction/Galois connection

property. The final lemmas show that, according to Figure 6, $[\![\langle A' \nleftarrow A\rangle]\!]$ is equivalent to the identity and $[\![\langle A'' \nleftarrow A'\rangle]\!][\![\langle A' \nleftarrow A\rangle]\!]$ is $[\![\langle A'' \nleftarrow A\rangle]\!]$, and similarly for downcasts. All of these properties are theorems in GTT (Section 3), and in the extended version it takes quite a bit of work to prove them true under translation, which illustrates that the axiomatic theory of GTT encodes a lot of information with relatively few rules.

As a corollary, we have the following conservativity result, which says that the homogeneous term dynamisms in GTT are sound and complete for inequalities in CBPV*.

Corollary 4.12 (Conservativity). *If $\Gamma \mid \Delta \vdash E, E' : T$ are two terms of the same type in the intersection of GTT and CBPV*, then $\Gamma \mid \Delta \vdash E \sqsubseteq E' : T$ is provable in GTT iff it is provable in CBPV*.*

Proof. The reverse direction holds because CBPV* is a syntactic subset of GTT. The forward direction holds by axiomatic graduality and the fact that identity casts are identities. □

5 COMPLEX VALUE/STACK ELIMINATION

Next, to bridge the gap between the semantic notion of complex value and stack with the more rigid operational notion, we perform a complexity-elimination pass. This translates a computation with complex values in it to an equivalent computation without complex values: i.e., all pattern matches take place in computations, rather than in values, and translates a term dynamism derivation that uses complex stacks to one that uses only "simple" stacks without pattern-matching and computation introduction forms. This translation clarifies the behavioral meaning of complex values and stacks, following Munch-Maccagnoni [2014]; Führmann [1999], and therefore of upcasts and downcasts.

Levy [2003] translates CBPV* to CBPV, but not does not prove the inequality preservation that we require here, so we give an alternative translation for which this property is easy to verify (see the extended version for full details). We translate both complex values and complex stacks to fully general computations, so that computation pattern-matching can replace the pattern-matching in complex values/stacks. More formally, we translate a CBPV* complex value $V : A$ to a CBPV computation $V^\dagger : \underline{F}A$ that in CBPV* is equivalent to $\mathtt{ret}\ V$. Similarly, we translate a CBPV* complex stack S with hole $\bullet : \underline{B}$ to a CBPV computation $S^\dagger$ with a free variable $z : U\underline{B}$ such that in CBPV*, $S^\dagger \sqsupseteq\sqsubseteq S[\mathtt{force}\ z]$. Computations $M : \underline{B}$ are translated to computations $M^\dagger$ with the same type.

Finally, we need to show that the translation preserves inequalities ($E^\dagger \sqsubseteq E'^\dagger$ if $E \sqsubseteq E'$), but because complex values and stacks satisfy more equations than arbitrary computations in the types of their translations do, we need to isolate the special "purity" property that their translations have. We show that complex values are translated to computations that satisfy *thunkability* [Munch-Maccagnoni 2014], which intuitively means M should have no observable effects, and so can be freely duplicated or discarded like a value. In the inequational theory of CBPV, this is defined by saying that running M to a value and then duplicating its value is the same as running M every time we need its value:

$$\Gamma \vdash \mathtt{ret}\ (\mathtt{thunk}\ M) \sqsupseteq\sqsubseteq \mathtt{bind}\ x \leftarrow M; \mathtt{ret}\ (\mathtt{thunk}\ (\mathtt{ret}\ x)) : \underline{F}U\underline{F}A$$

Dually, we show that complex stacks are translated to computations that satisfy (semantic) *linearity* [Munch-Maccagnoni 2014], where intuitively a computation M with a free variable $x : U\underline{B}$ is linear in x if M behaves as if when it is forced, the first thing it does is forces x, and that is the only time it uses x. This is described in the CBPV inequational theory as follows:

$$\Gamma, z : U\underline{F}U\underline{B} \vdash \mathtt{bind}\ x \leftarrow \mathtt{force}\ z; M \sqsupseteq\sqsubseteq M[\mathtt{thunk}\ (\mathtt{bind}\ x \leftarrow (\mathtt{force}\ z); \mathtt{force}\ x)]$$

Composing this with the translation from GTT to CBPV* shows that *GTT value upcasts are thunkable and computation downcasts are linear*, which justifies a number of program transformations.

6 OPERATIONAL MODEL OF GTT

In this section, we establish a model of our CBPV inequational theory using a notion of observational approximation based on the CBPV operational semantics. By composition with the axiomatic graduality theorem, this establishes the *operational graduality* theorem, i.e., a theorem analogous to the *dynamic gradual guarantee* [Siek et al. 2015a].

6.1 Call-by-push-value operational semantics

We use a small-step operational semantics for CBPV with the following rules (excerpt):

$$\begin{array}{rcl}
S[\mho] & \mapsto^0 & \mho \\
S[\texttt{split}\ (V_1, V_2)\ \texttt{to}\ (x_1, x_2).M] & \mapsto^0 & S[M[V_1/x_1, V_2/x_2]] \\
S[\texttt{unroll}\ \texttt{roll}_A\ V\ \texttt{to}\ \texttt{roll}\ x.M] & \mapsto^1 & S[M[V/x]] \\
S[\texttt{force}\ \texttt{thunk}\ M] & \mapsto^0 & S[M] \\
S[\texttt{bind}\ x \leftarrow \texttt{ret}\ V; M] & \mapsto^0 & S[M[V/x]] \\
S[(\lambda x : A.M)\, V] & \mapsto^0 & S[M[V/x]] \\
S[\texttt{unroll}\ \texttt{roll}_{\underline{B}}\ M] & \mapsto^1 & S[M]
\end{array}$$

$$\frac{}{M \Mapsto^0 M} \qquad \frac{M_1 \mapsto^i M_2 \qquad M_2 \Mapsto^j M_3}{M_1 \Mapsto^{i+j} M_3}$$

This is morally the same as in Levy [2003], but we present stacks in a manner similar to Hieb-Felleisen style evaluation contexts. We also make the step relation count unrollings of a recursive or corecursive type, for the step-indexed logical relation later. The operational semantics is only defined for terms of type $\cdot \vdash M : \underline{F}(1+1)$, which we take as the type of whole programs.

It is easy to see that the operational semantics is deterministic and progress and type preservation theorems hold, which allows us to define the "final result" of a computation as follows:

Corollary 6.1 (Possible Results of Computation). *For any* $\cdot \vdash M : \underline{F}2$, *either* $M \Uparrow$ *or* $M \Downarrow \mho$ *or* $M \Downarrow \texttt{ret true}$ *or* $M \Downarrow \texttt{ret false}$.

Definition 6.2 (Results). The possible results of a computation are $\Omega, \mho, \texttt{ret true}$ and $\texttt{ret false}$. We denote a result by R, and define a function result which takes a program $\cdot \vdash M : \underline{F}2$, and returns its end-behavior, i.e., $\text{result}(M) = \Omega$ if $M \Uparrow$ and otherwise $M \Downarrow \text{result}(M)$.

6.2 Observational Equivalence and Approximation

Next, we define observational equivalence and approximation in CBPV. Define a context C to be a term/value/stack with a single $[\cdot]$ as some subterm/value/stack, and define a typing $C : (\Gamma \vdash \underline{B}) \Rightarrow (\Gamma' \vdash \underline{B}')$ to hold when for any $\Gamma \vdash M : \underline{B}$, $\Gamma' \vdash C[M] : \underline{B}'$ (and similarly for values/stacks). Using contexts, we can lift any relation on *results* to relations on open terms, values and stacks.

Definition 6.3 (Contextual Lifting). Given any relation $\sim \subseteq \text{Result}^2$, we can define its *observational lift* $\sim^{\text{ctx}}$ to be the typed relation defined by

$$\Gamma \mid \Delta \vDash E \sim^{\text{ctx}} E' \in T = \forall C : (\Gamma \mid \Delta \vdash T) \Rightarrow (\cdot \vdash \underline{F}2).\ \text{result}(C[E]) \sim \text{result}(C[E'])$$

The contextual lifting $\sim^{\text{ctx}}$ is a preorder or equivalence relation whenever the original relation $\sim$ is, and all $\sim$'s we use will be at least preorders, so we write $\trianglelefteq$ instead of $\sim$ for a relation on results. Three important relations arise as liftings: Equality of results lifts to observational equivalence ($=^{\text{ctx}}$). The preorder generated by $\mho \sqsubseteq R$ (i.e. the other three results are unrelated maximal elements) lifts to the notion of *error approximation* used in New and Ahmed [2018] to prove the graduality property ($\sqsubseteq^{\text{ctx}}$). The preorder generated by $\Omega \preceq R$ lifts to the standard notion of *divergence approximation* ($\preceq^{\text{ctx}}$).

The goal of this section is to prove that a symmetric equality $E \sqsupseteq\sqsubseteq E'$ in CBPV (i.e. $E \sqsubseteq E'$ and $E' \sqsubseteq E$) implies contextual equivalence $E =^{\text{ctx}} E'$ and that inequality in CBPV $E \sqsubseteq E'$ implies error approximation $E \sqsubseteq^{\text{ctx}} E'$, proving graduality of the operational model.Because we have

$$
\begin{array}{rcl}
V_1 \trianglelefteq^{\log}_{0,i} V_2 & = & \bot \\
(V_1, V_1') \trianglelefteq^{\log}_{A\times A',i} (V_2, V_2') & = & V_1 \trianglelefteq^{\log}_{A,i} V_2 \wedge V_1' \trianglelefteq^{\log}_{A',i} V_2' \\
\mathtt{roll}_{\mu X.A}\ V_1 \trianglelefteq^{\log}_{\mu X.A,i} \mathtt{roll}_{\mu X.A}\ V_2 & = & i = 0 \vee V_1 \trianglelefteq^{\log}_{A[\mu X.A/X],i-1} V_2 \\
V_1 \trianglelefteq^{\log}_{U\underline{B},i} V_2 & = & \forall j \le i, S_1 \trianglelefteq^{\log}_{\underline{B},j} S_2.\ S_1[\mathtt{force}\ V_1] \trianglelefteq^{j} \mathrm{result}(S_2[\mathtt{force}\ V_2]) \\
\\
S_1[\bullet V_1] \trianglelefteq^{\log}_{A\to\underline{B},i} S_1[\bullet V_2] & = & V_1 \trianglelefteq^{\log}_{A,i} V_2 \wedge S_1 \trianglelefteq^{\log}_{\underline{B},i} S_2 \\
S_1[\mathtt{unroll}\ \bullet] \trianglelefteq^{\log}_{\nu\underline{Y}.\underline{B},i} S_2[\mathtt{unroll}\ \bullet] & = & i = 0 \vee S_1 \trianglelefteq^{\log}_{\underline{B}[\nu\underline{Y}.\underline{B}/\underline{Y}],i-1} S_2 \\
S_1 \trianglelefteq^{\log}_{\underline{F}A,i} S_2 & = & \forall j \le i, V_1 \trianglelefteq^{\log}_{A,j} V_2.\ S_1[\mathtt{ret}\ V_1] \trianglelefteq^{j} \mathrm{result}(S_2[\mathtt{ret}\ V_2])
\end{array}
$$

Fig. 7. Logical Relation from a Preorder $\trianglelefteq$ (selected cases)

non-well-founded μ/ν types, we use a *step-indexed logical relation* to prove properties about the contextual lifting of certain preorders $\trianglelefteq$ on results. In step-indexing, the *infinitary* relation given by $\trianglelefteq^{\mathrm{ctx}}$ is related to the set of all of its *finitary approximations* $\trianglelefteq^{i}$, which "time out" after observing i steps of evaluation and declare that the terms *are* related. A preorder $\trianglelefteq$ is only recoverable from its finite approximations if Ω is a *least* element, $\Omega \trianglelefteq R$, because a diverging term will cause a time out for any finite index. We call a preorder with $\Omega \trianglelefteq R$ a *divergence preorder*. But this presents a problem, because *neither* of our intended relations ($=$ and $\sqsubseteq$) is a divergence preorder; rather both have Ω as a *maximal* element. For observational equivalence, because contextual equivalence is symmetric divergence approximation ($M =^{\mathrm{ctx}} N$ iff $M \preceq^{\mathrm{ctx}} N$ and $N \preceq^{\mathrm{ctx}} M$), we can use a step-indexed logical relation to characterize $\preceq$, and then obtain results about observational equivalence from that [Ahmed 2006]. A similar move works for error approximation [New and Ahmed 2018], but since $R \sqsubseteq R'$ is *not* symmetric, it is decomposed as the conjunction of two orderings: error approximation up to divergence on the left $\preceq\!\sqsubseteq$ (the preorder where $\mathcal{U}$ and Ω are both minimal: $\mathcal{U} \preceq\!\sqsubseteq R$ and $\Omega \preceq\!\sqsubseteq R$) and error approximation up to divergence on the right $\sqsubseteq\!\succeq$ (the diamond preorder where $\mathcal{U}$ is minimal and Ω is maximal, with $\mathtt{true}/\mathtt{false}$ in between). Then $\preceq\!\sqsubseteq$ and the *opposite* of $\sqsubseteq\!\succeq$ (written $\preceq\!\sqsupseteq$) are divergence preorders, so we can use a step-indexed logical relation to characterize them. Overall, because $=$ is symmetric $\sqsubseteq$, and $\sqsubseteq$ is the conjunction of $\preceq\!\sqsubseteq$ and $\sqsubseteq\!\succeq$, and contextual lifting commutes with conjunction and opposites, it will suffice to develop logical relations for divergence preorders.

6.3 CBPV Step Indexed Logical Relation

We use a logical relation to prove results about $E \trianglelefteq^{\mathrm{ctx}} E'$ where $\trianglelefteq$ is a divergence preorder. The "finitization" of a divergence preorder is a relation between *programs* and *results*: a program approximates a result R at index i if it reduces to R in $< i$ steps or it "times out" by reducing at least i times.

Definition 6.4 (Finitized Preorder). Given a divergence preorder $\trianglelefteq$, we define the *finitization* of $\trianglelefteq$ to be, for each natural number i, a relation between programs and results defined by

$$M \trianglelefteq^{i} R = (\exists M'.\ M \mapsto^{i} M') \vee (\exists (j < i). \exists R_M.\ M \mapsto^{j} R_M \wedge R_M \trianglelefteq R)$$

The (closed) *logical* preorder (for closed values/stacks) is in Figure 7. For every i and value type A, we define a relation $\trianglelefteq^{\log}_{A,i}$ between two closed values of type A, and for every i and $\underline{B}$, we define a relation for two "closed" stacks $\underline{B} \vdash \underline{F}2$ outputting the observation type $\underline{F}2$—the definition is by mutual lexicographic induction on i and $A/\underline{B}$. Two values or stacks are related if they have the same structure, where for μ, ν we decrement i and succeed if $i = 0$. The shifts $\underline{F}/U$ take the *orthogonal* of

the relation: the set of all stacks/values that when composed with those values/stacks are related by $\preceq^{j \leq i}$; the quantifier over $j \leq i$ is needed to make the relation downward closed.

The logical preorder for open terms is defined as usual by quantifying over all related closing substitutions, but also over all stacks to the observation type $\underline{F}(1+1)$:

Definition 6.5 (Logical Preorder). For a divergence preorder $\preceq$, its step-indexed logical preorder is for terms (open stack, value cases are defined in the extended version): $\Gamma \vDash M_1 \preceq_i^{\log} M_2 \in \underline{B}$ iff for every $\gamma_1 \preceq^{\log}_{\Gamma,i} \gamma_2$ and $S_1 \preceq^{\log}_{\underline{B},i} S_2$, $S_1[M_1[\gamma_1]] \preceq^i \text{result}(S_2[M_2[\gamma_2]])$.

Next, we show the fundamental theorem:

Theorem 6.6 (Logical Preorder is a Congruence). *For any divergence preorder, the logical preorder $E \preceq_i^{log} E'$ is closed under applying any value/term/stack constructors to both sides.*

This in particular implies that the relation is reflexive ($\Gamma \vDash M \preceq_i^{\log} M \in \underline{B}$ for all well-typed M),so we have the following *strengthening* of the progress-and-preservation type soundness theorem: because $\preceq^i$ only counts unrolling steps, terms that never use μ or ν types (for example) are guaranteed to terminate.

Corollary 6.7 (Unary LR). *For every program $\cdot \vdash M : \underline{F}2$ and $i \in \mathbb{N}$, $M \preceq^i \mathit{result}(M)$*

Using reflexivity, we prove that the indexed relation between terms and results recovers the original preorder in the limit as $i \to \omega$. We write $\preceq^\omega$ to mean the relation holds for every i, i.e., $\preceq^\omega = \bigcap_{i \in \mathbb{N}} \preceq^i$.

Corollary 6.8 (Limit Lemma). *For any divergence preorder $\preceq$, $\mathit{result}(M) \preceq R$ iff $M \preceq^\omega R$.*

Corollary 6.9 (Logical implies Contextual). *If $\Gamma \vDash E \preceq^{log}_\omega E' \in \underline{B}$ then $\Gamma \vDash E \preceq^{ctx} E' \in \underline{B}$.*

Proof. Let C be a closing context. By congruence, $C[M] \preceq^{\log}_\omega C[N]$, so using empty environment and stack, $C[M] \preceq^\omega \text{result}(C[N])$ and by the limit lemma, we have $\text{result}(C[M]) \preceq \text{result}(C[N])$. □

This establishes that our logical relation can prove graduality, so it only remains to show that our *inequational theory* implies our logical relation. Having already validated the congruence rules and reflexivity, we validate the remaining rules of transitivity, error, substitution, and $\beta\eta$ for each type constructor. Other than the $\mho \sqsubseteq M$ rule, all of these hold for any divergence preorder.

For transitivity, with the unary model and limiting lemmas in hand, we can prove that all of our logical relations (open and closed) are transitive in the limit. To do this, we first prove the following kind of "quantitative" transitivity lemma, and then transitivity in the limit is a consequence.

Lemma 6.10 (Logical Relation is Quantitatively Transitive).
If $V_1 \preceq^{log}_{A,i} V_2$ and $V_2 \preceq^{log}_{A,\omega} V_3$, then $V_1 \preceq^{log}_{A,i} V_3$, and analogously for stacks.

Corollary 6.11 (Logical Relation is Transitive in the Limit). *$\preceq^{log}_\omega$ is transitive.*

For errors, the strictness axioms hold for any $\preceq$, but the axiom that $\mho$ is a least element is specific to the definitions of $\preceq\sqsubseteq, \sqsubseteq\succeq$

Lemma 6.12 (Error Rules). *For any divergence preorder $\preceq$ and appropriately typed S, M,*

$$S[\mho] \preceq^{log}_\omega \mho \qquad \mho \preceq^{log}_\omega S[\mho] \qquad \mho \preceq\sqsubseteq^{log}_\omega M \qquad M \preceq\sqsupseteq^{log}_\omega \mho$$

The lemmas we have proved cover all of the inequality rules of CBPV, so applying them with $\preceq$ chosen to be $\preceq\sqsubseteq$ and $\preceq\sqsupseteq$ gives

Lemma 6.13 ($\preceq\sqsubseteq$ and $\sqsubseteq\succeq$ are Models of CBPV). *If* $\Gamma \mid \Delta \vdash E \sqsubseteq E' : \underline{B}$ *then* $\Gamma \mid \Delta \vDash E \preceq\sqsubseteq^{\omega} E' \in \underline{B}$ *and* $\Gamma \mid \Delta \vDash E' \preceq\sqsupseteq^{\omega} E \in \underline{B}$.

Because logical implies contextual equivalence, we can conclude with the main theorem:

Theorem 6.14 (Contextual Approximation/Equivalence Model CBPV).
If $\Gamma \mid \Delta \vdash E \sqsubseteq E' : T$ *then* $\Gamma \mid \Delta \vDash E \sqsubseteq^{ctx} E' \in T$; *if* $\Gamma \mid \Delta \vdash E \sqsupseteq\sqsubseteq E' : T$ *then* $\Gamma \mid \Delta \vDash E =^{ctx} E' \in T$.

7 DISCUSSION AND RELATED WORK

In this paper, we have given a logic for reasoning about gradual programs in a mixed call-by-value/call-by-name language, shown that the axioms uniquely determine almost all of the contract translation implementing runtime casts, and shown that the axiomatics is sound for contextual equivalence/approximation in an operational model. In immediate future work, we believe it is straightforward to add inductive/coinductive types and obtain similar unique cast implementation theorems (e.g. $\langle \texttt{list}(A') \nwarrow \texttt{list}(A) \rangle \sqsupseteq\sqsubseteq \texttt{map}\langle A' \nwarrow A \rangle$). Additionally, since more efficient cast implementations such as optimized cast calculi (the lazy variant in Herman et al. [2010]) and threesome casts [Siek and Wadler 2010], are equivalent to the lazy contract semantics, they should also be models of GTT, and if so we could use GTT to reason about program transformations and optimizations in them.

The cast uniqueness principles given in theorem 3.5 are theorems in the formal logic of Gradual Type Theory, and so there is a question of to what languages the theorem applies. The theorem applies to any *model* of gradual type theory, such as the models we have constructed using call-by-push-value given in Sections 4, 5, 6. We conjecture that simple call-by-value and call-by-name gradual languages are also models of GTT, by extending the translation of call-by-push-value into call-by-value and call-by-name in the appendix of Levy's monograph [Levy 2003]. In order for the theorem to apply, the language must validate an appropriate version of the η principles for the types. So for example, a call-by-value language that has reference equality of functions does *not* validate even the value-restricted η law for functions, and so the case for functions does not apply. It is a well-known issue that in the presence of pointer equality of functions, the lazy semantics of function casts is not compatible with the graduality property, and our uniqueness theorem provides a different perspective on this phenomenon [Findler et al. 2004; Strickland et al. 2012; Siek et al. 2015a]. However, we note that the cases of the uniqueness theorem for each type connective are completely *modular*: they rely only on the specification of casts and the β, η principles for the particular connective, and not on the presence of any other types, even the dynamic types. So even if a call-by-value language may have reference equality functions, if it has the η principle for strict pairs, then the pair cast must be that of Theorem 3.5.

Next, we consider the applicability to non-eager languages. Analogous to call-by-value, our uniqueness principle should apply to simple *call-by-name* gradual languages, where full η equality for functions is satisfied, but η equality for booleans and strict pairs requires a "stack restriction" dual to the value restriction for call-by-value function η. We are not aware of any call-by-name gradual languages, but there is considerable work on *contracts* for non-eager languages, especially Haskell [Hinze et al. 2006; Xu et al. 2009]. However, we note that Haskell is *not* a call-by-name language in our sense for two reasons. First, Haskell uses call-by-need evaluation where results of computations are memoized. However, when only considering Haskell's effects (error and divergence), this difference is not observable so this is not the main obstacle. The bigger difference between Haskell and call-by-name is that Haskell supports a `seq` operation that enables the programmer to force evaluation of a term to a value. This means Haskell violates the function η principle because Ω will cause divergence under `seq`, whereas $\lambda x.\Omega$ will not. This is a crucial feature of Haskell and is a major source of differences between implementations of lazy contracts, as noted in Degen et al. [2012].

We can understand this difference by using a different translation into call-by-push-value: what Levy calls the "lazy paradigm", as opposed to call-by-name [Levy 2003]. Simply put, connectives are interpreted as in call-by-value, but with the addition of extra thunks UF, so for instance the lazy function type $A \to B$ is interpreted as $UFU(UFA \to FB)$ and the extra UFU here is what causes the failure of the call-by-name η principle. With this embedding and the uniqueness theorem, GTT produces a definition for lazy casts, and the definition matches the work of Xu et al. [2009] when restricting to non-dependent contracts.

Greenman and Felleisen [2018] gives a spectrum of differing syntactic type soundness theorems for different semantics of gradual typing. Our work here is complementary, showing that certain program equivalences can only be achieved by certain cast semantics.

Degen et al. [2012] give an analysis of different cast semantics for contracts in lazy languages, specifically based on Haskell, i.e., call-by-need with seq. They propose two properties "meaning preservation" and "completeness" that they show are incompatible and identify which contract semantics for a lazy language satisfy which of the properties. The meaning preservation property is closely related to graduality: it says that evaluating a term with a contract either produces blame or has the same observable effect as running the term without the contract. Meaning preservation rules out overly strict contract systems that force (possibly diverging) thunks that wouldn't be forced in a non-contracted term. Completeness, on the other hand, requires that when a contract is attached to a value that it is *deeply* checked. The two properties are incompatible because, for instance, a pair of a diverging term and a value can't be deeply checked without causing the entire program to diverge. Using Levy's embedding of the lazy paradigm into call-by-push-value their incompatibility theorem should be a consequence of our main theorem in the following sense. We showed that any contract semantics departing from the implementation in Theorem 3.5 must violate η or graduality. Their completeness property is inherently eager, and so must be different from the semantics GTT would provide, so either the restricted η or graduality fails. However, since they are defining contracts within the language, they satisfy the restricted η principle provided by the language, and so it must be graduality, and therefore meaning preservation that fails.

Henglein's work on dynamic typing also uses an axiomatic semantics of casts, but axiomatizes behavior of casts at each type directly whereas we give a uniform definition of all casts and derive implementations for each type [Henglein 1994]. Because of this, the theorems proven in that paper are more closely related to our model construction in Section 4. More specifically, many of the properties of casts needed to prove Theorem 4.11 have direct analogues in Henglein's work, such as the coherence theorems. We have not included these lemmas in the paper because they are quite similar to lemmas proven in New and Ahmed [2018]; see there for a more detailed comparison, and the extended version of this paper for full proof details [New et al. 2018]. Finally, we note that our assumption of compositionality, i.e., that all casts can be decomposed into an upcast followed by a downcast, is based on Henglein's analysis, where it was proven to hold in his coercion calculus.

In this work we have applied a method of "gradualizing" axiomatic type theories by adding in dynamism orderings and adding dynamic types, casts and errors by axioms related to the dynamism orderings. This is similar in spirit to two recent frameworks for designing gradual languages: Abstracting Gradual Typing (AGT) [Garcia et al. 2016] and the Gradualizer [Cimini and Siek 2016, 2017]. All of these approaches start with a typed language and construct a related gradual language. A major difference between our approach and those is that our work is based on axiomatic semantics and so we take into account the equality principles of the typed language, whereas Gradualizer is based on the typing and operational semantics and AGT is based on the type safety proof of the typed language. Furthermore, our approach produces not just a single language, but also an axiomatization of the structure of gradual typing and so we can prove results about many languages by proving theorems in GTT. The downside to this is that our approach

doesn't directly provide an operational semantics for the gradual language, whereas for AGT this is a semi-mechanical process and for Gradualizer, completely automated. Finally, we note that AGT produces the "eager" semantics for function types, and it is not clear how to modify the AGT methodology to reproduce the lazy semantics that GTT provides. More generally, both AGT and the Gradualizer are known to produce violations of parametricity when applied to polymorphic languages, with the explanation being that the parametricity property is in no way encoded in the input to the systems: the operational semantics and the type safety proof. In future work, we plan to apply our axiomatic approach to gradualizing polymorphism and state by starting with the rich *relational logics and models* of program equivalence for these features [Plotkin and Abadi 1993; Dunphy 2002; Matthews and Ahmed 2008; Neis et al. 2009; Ahmed et al. 2009], which may lend insight into existing proposals [Siek et al. 2015b; Ahmed et al. 2017; Igarashi et al. 2017a; Siek and Taha 2006]— for example, whether the "monotonic" [Siek et al. 2015b] and "proxied" [Siek and Taha 2006] semantics of references support relational reasoning principles of local state.

We do not give a treatment of runtime blame reporting, but we argue that the observation that upcasts are thunkable and downcasts are linear is directly related to blame soundness [Tobin-Hochstadt and Felleisen 2006; Wadler and Findler 2009] in that if an upcast were *not* thunkable, it should raise positive blame and if a downcast were *not* linear, it should raise negative blame. First, consider a potentially effectful stack upcast of the form $\langle \underline{F}A' \nwarrow \underline{F}A \rangle$. If it is not thunkable, then in our logical relation this would mean there is a value $V : A$ such that $\langle \underline{F}A' \nwarrow \underline{F}A \rangle(\mathtt{ret}\ V)$ performs some effect. Since the only observable effects for casts are dynamic type errors, $\langle \underline{F}A' \nwarrow \underline{F}A \rangle(\mathtt{ret}\ V) \mapsto \mathfrak{U}$, and we must decide whether the positive party or negative party is at fault. However, since this is call-by-value evaluation, this error happens unconditionally on the continuation, so the continuation never had a chance to behave in such a way as to prevent blame, and so we must blame the positive party. Dually, consider a value downcast of the form $\langle U\underline{B} \swarrow U\underline{B}' \rangle$. If it is not linear, that would mean it forces its $U\underline{B}'$ input either never or more than once. Since downcasts should refine their inputs, it is not possible for the downcast to use the argument twice, since e.g. printing twice does not refine printing once. So if the cast is not linear, that means it fails without ever forcing its input, in which case it knows nothing about the positive party and so must blame the negative party. In future work, we plan to investigate extensions of GTT with more than one $\mathfrak{U}$ with different blame labels, and an axiomatic account of a blame-aware observational equivalence.

Gradual session types [Igarashi et al. 2017b] share some similarities to GTT, in that there are two sorts of types (values and sessions) with a dynamic value type and a dynamic session type. However, their language is not *polarized* in the same way as CBPV, so there is not likely an analogue between our upcasts always being between value types and downcasts always being between computation types. Instead, we might reconstruct this in a polarized session type language [Pfenning and Griffith 2015].

We also plan to explore using GTT's specification of casts in a dependently typed setting, building on work using Galois connections for casts between dependent types [Dagand et al. 2018], and work on effectful dependent types based a CBPV-like judgement structure [Ahman et al. 2016].

Acknowledgments. We thank Ron Garcia, Kenji Maillard and Gabriel Scherer for helpful discussions about this work. We thank the anonymous reviewers for helpful feedback on this article. This material is based on research sponsored by the National Science Foundation under grant CCF-1453796 and the United States Air Force Research Laboratory under agreement number FA9550-15-1-0053 and FA9550-16-1-0292. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government, or Carnegie Mellon University.

REFERENCES

Danel Ahman, Neil Ghani, and Gordon D. Plotkin. 2016. Dependent Types and Fibred Computational Effects. In *Foundations of Software Science and Computation Structures*. 36–54.

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *European Symposium on Programming (ESOP)*. 69–83.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-Dependent Representation Independence. In *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom*.

Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

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 (POPL '16)*.

Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. 789–803.

Pierre-Èvariste Dagand, Nicolas Tabareau, and Èric Tanter. 2018. Foundations of dependent interoperability. *Journal of Functional Programming* 28 (2018), e9. https://doi.org/10.1017/S0956796818000011

Markus Degen, Peter Thiemann, and Stefan Wehr. 2012. The interaction of contracts and laziness. *Higher-Order and Symbolic Computation* 25 (2012), 85–125.

Brian Patrick Dunphy. 2002. *Parametricity As a Notion of Uniformity in Reflexive Graphs*. Ph.D. Dissertation. Champaign, IL, USA. Advisor(s) Reddy, Uday.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*. 48–59.

Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. 2004. Semantic Casts: Contracts and Structural Subtyping in a Nominal World. In *European Conference on Object-Oriented Programming (ECOOP)*.

Carsten Führmann. 1999. Direct models of the computational lambda-calculus. *Electronic Notes in Theoretical Computer Science* 20 (1999), 245–292.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *ACM Symposium on Principles of Programming Languages (POPL)*.

Jean-Yves Girard. 2001. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science* 11, 3 (2001), 301–506.

Michael Greenberg. 2015. Space-Efficient Manifest Contracts. In *ACM Symposium on Principles of Programming Languages (POPL)*. 181–194.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest *(POPL '10)*.

Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. In *International Conference on Functional Programming (ICFP), St. Louis, Missouri*.

Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. 22, 3 (1994), 197–230.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* (2010).

Ralf Hinze, Johan Jeuring, and Andres Löh. 2006. Typed Contracts for Functional Programming. In *International Symposium on Functional and Logic Programming (FLOPS)*.

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017b. Gradual Session Types. *Proceedings of ACM Programning Languages* 1, ICFP, Article 38 (Aug. 2017), 28 pages.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017a. On Polymorphic Gradual Typing. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom*.

Paul Blain Levy. 2003. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Springer.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 500–514.

Jacob Matthews and Amal Ahmed. 2008. Parametric polymorphism through run-time sealing, or, Theorems for low, low prices!. In *European Symposium on Programming (ESOP)*.

Guillaume Munch-Maccagnoni. 2014. Models of a Non-associative Composition. In *Foundations of Software Science and Computation Structures*. 396–410.

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-Parametric Parametricity. In *International Conference on Functional Programming (ICFP)*. 135–148.

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. In *International Conference on Functional Programming (ICFP), St. Louis, Missouri*.

Max S. New and Daniel R. Licata. 2018. Call-by-name Gradual Type Theory. *FSCD* (2018).

Max S. New, Daniel R. Licata, and Amal Ahmed. 2018. Gradual Type Theory (Extend Version). (2018). https://arxiv.org/abs/1811.02440

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types (invited talk). In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*.

Gordon D. Plotkin and Martín Abadi. 1993. A Logic for Parametric Polymorphism. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*. 361–375.

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *European Symposium on Programming (ESOP)*. Springer-Verlag, Berlin, Heidelberg, 17–31.

Jeremy Siek and Sam Tobin-Hochstadt. 2016. The recursive union of some gradual types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Springer LNCS)* volume 9600 (2016).

Jeremy Siek, Micahel Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop (Scheme)*. 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and Without Blame. In *ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 365–376.

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and Impersonators: Run-time Support for Reasonable Interposition *(ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA))*.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium (DLS)*. 964–974.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems *(POPL 2017)*.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*. 1–16.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. Static Contract Checking for Haskell *(ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia)*.

Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching*. Ph.D. Dissertation. Carnegie Mellon University.