Graduality from Embedding-Projection Pairs[*]

MAX S. NEW, Northeastern University, USA
AMAL AHMED, Northeastern University, USA and Inria Paris, France

Gradually typed languages allow statically typed and dynamically typed code to interact while maintaining benefits of both styles. The key to reasoning about these mixed programs is Siek-Vitousek-Cimini-Boyland's (dynamic) gradual guarantee, which says that giving components of a program more precise types only adds runtime type checking, and does not otherwise change behavior. In this paper, we give a semantic reformulation of the gradual guarantee called *graduality*. We change the name to promote the analogy that graduality is to gradual typing what parametricity is to polymorphism. Each gives a local-to-global, syntactic-to-semantic reasoning principle that is formulated in terms of a kind of observational approximation.

Utilizing the analogy, we develop a novel logical relation for proving graduality. We show that *embedding-projection pairs (ep pairs)* are to graduality what relations are to parametricity. We argue that casts between two types where one is "more dynamic" (less precise) than the other necessarily form an ep pair, and we use this to cleanly prove the graduality cases for casts from the ep-pair property. To construct ep pairs, we give an analysis of the *type dynamism* relation—also known as type precision or naïve subtyping—that interprets the rules for type dynamism as compositional constructions on ep pairs, analogous to the coercion interpretation of subtyping.

CCS Concepts: • **Theory of computation** → **Semantics and reasoning**; • **Software and its engineering** → **Formal language definitions**; *Functional languages*;

Additional Key Words and Phrases: gradual typing, dynamic gradual guarantee, observational error approximation, logical relations

ACM Reference Format:
Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proc. ACM Program. Lang.* 2, ICFP, Article 73 (September 2018), 30 pages. https://doi.org/10.1145/3236768

1 INTRODUCTION

Gradually typed programming languages are designed to resolve the conflict between static and dynamically typed programming styles [Tobin-Hochstadt and Felleisen 2006, 2008; Siek and Taha 2006]. A gradual language allows a smooth transition from dynamic to static typing through gradual addition of types to dynamically typed programs, and allows for safe interactions between more statically typed and more dynamically typed components. With such an enticing goal there has been extensive research on gradual typing—e.g., [Siek and Taha 2006; Gronski et al. 2006; Wadler and Findler 2009; Ina and Igarashi 2011; Swamy et al. 2014; Allende et al. 2013]—with recent work aimed at extending gradual typing to more advanced language features, such as parametric polymorphism [Ahmed et al. 2017; Igarashi et al. 2017a], effect tracking [Bañados Schwerter

[*]In this paper, we use blue to typeset our gradual cast calculus λ_G and **red** to typeset our typed language with errors $\lambda_{T,\mho}$. The paper will be much easier to read if viewed/printed in color.

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

2475-1421/2018/9-ART73
https://doi.org/10.1145/3236768

et al. 2014], typestate [Wolff et al. 2011], session types [Igarashi et al. 2017b], and refinement types [Lehmann and Tanter 2017].

Formalizing the idea of a "smooth transition", a key property that every gradually typed language should satisfy is Siek, Vitousek, Cimini, and Boyland's *(dynamic)*[1] *gradual guarantee*, which we refer to as *graduality* (by analogy with parametricity). Graduality enables programmers to modify their program from a dynamically typed to a statically typed style, and vice-versa, with confidence that the program's behavior only changes in predictable ways. Specifically, it says that changing the types in a program to be "less dynamic"/"more precise"—i.e., changing from the dynamic type to some more precise type such as integers or functions—either produces the same behavior as the original program or causes a dynamic type error. Conversely, if a program does not error and some types are made "more dynamic"/"less precise" then the program has the exact same behavior. This is an important reasoning principle for programmers as the alternative would be quite counterintuitive: for instance, changing certain type annotations might cause a terminating program to diverge, or a program that prints your calendar to tweet your home address! This distinguishes dynamic type checking in gradual typing from exceptions: raising an exception is a valid program behavior that can be caught and handled by a caller, whereas a dynamic type error is always considered to be a bug, and terminates the program.

More formally, the notion of when a type A is "less dynamic" than another type B is specified by a *type dynamism* relation (also known as type precision or naïve subtyping), written $A \sqsubseteq B$, which is defined for simple languages as the least congruence relation such that the dynamic type $?$ is the *most* dynamic type: $A \sqsubseteq ?$. Then, *term dynamism* (also known as term precision) is the natural extension of type dynamism to terms, written $t \sqsubseteq s$. The graduality theorem is then that if $t \sqsubseteq s$, then the behavior of t must be "less dynamic" than the behavior of s—that is, either t produces a runtime type error or both terms have the exact same behavior. We say t is "less dynamic" in the sense that it has *fewer behaviors.*

Unfortunately, for the majority of gradually typed languages, the (dynamic) gradual guarantee is considered quite challenging to prove, and there is only limited guidance about how to design new languages so that they satisfy this property. There are two notable exceptions: Abstracting Gradual Typing (AGT) [Garcia et al. 2016a] and the Gradualizer [Cimini and Siek 2016, 2017] provide systematic methods and formal tools, respectively, for deriving a gradually typed language from a statically typed language, and they both provide the gradual guarantee by construction. However, while they provide a proof of the gradual guarantee for languages produced in the respective frameworks, most gradually typed languages are not produced in this way; for instance, Typed Racket's approach to gradual typing [Tobin-Hochstadt and Felleisen 2006, 2008] is not explained by either system. Furthermore, both Gradualizer and AGT base their semantics on static type checking itself, but this is the reverse of the semantic view of type checking. In the semantic viewpoint, type checking should be justified by a sensible semantics, and not the other way around.

Type Dynamism and Embedding-Projection Pairs. While the gradual guarantee as presented in Siek et al. [2015] makes type dynamism a central component, the semantic meaning of type dynamism is unclear. This is not just a philosophical question: it is unclear how to extend type dynamism to new language features. For instance, polymorphic gradually typed languages have been developed recently by Ahmed et al. [2017] and Igarashi et al. [2017a], but the two papers have different definitions of type dynamism, and neither attempts a proof of the (dynamic) gradual guarantee. The AGT [Garcia et al. 2016a] approach gives a systematic definition of type dynamism in terms

[1]The same work also introduces a *static* gradual guarantee that says that changing the types in a program to be less dynamic means type checking becomes stricter. We do not consider this in our paper because we only consider the semantics of cast calculi, not the type systems of gradual surface languages. We discuss the relationship further in §7

of sets of static types, but that definition is difficult to separate from the rest of their framework, whereas we would like a definition that can be interpreted in any gradually typed language. At present, the best guidance we have comes from the gradual guarantee itself: the dynamic type should be the greatest element, and the gradual guarantee should hold.

We propose a semantic definition for type dynamism that naturally leads to a clean formulation and proof of the gradual guarantee: An ordering $A \sqsubseteq B$ should hold when the casts between the two types form an *embedding-projection pair.*

What does this mean? First, in order to support interaction between statically typed and dynamic typed code while still maintaining the guarantees of the static types, gradually typed languages include *casts*[2] $\langle B \Leftarrow A\rangle$ that dynamically check if a value of type A corresponds to a valid inhabitant of the type B, and if so, transform its value to have the right type. Then if $A \sqsubseteq B$, we say that the casts $\langle B \Leftarrow A\rangle$ and $\langle A \Leftarrow B\rangle$ form an *embedding-projection pair*, which means that they satisfy the following two properties that describe acceptable behaviors when casting between the two types: *retraction* and *projection.*

First, A should be a *stricter* type than B, so anything satisfying A should also satisfy B. This is captured in the *retraction* property: if we cast a value $v : A$ from A to B and then back down to A, we should get back an equivalent value because v should satisfy the type of A and B. Formally, $\langle A \Leftarrow B\rangle\langle B \Leftarrow A\rangle t \approx t$ where $\approx$ means *observational equivalence* of the programs: when placed in the same spot in a program, they produce the same behavior.

Second, casts should only be doing type *checking*, and not otherwise changing the behavior of the term. Since B is a weaker property than A, if we cast a value of $v : B$ down to A, there may be a runtime type error. However, if v really does satisfy A the cast succeeds, and if we cast back to B we should get back a value with similar behavior to v. If B is a first-order type like booleans or numbers, we should get back exactly the same value. However, if A, B are higher-order types like functions or objects, then it is impossible to check if a value of one type B satisfies A. For instance, if $B = ? \to ?$ and $A = ? \to \mathbb{N}$, then it is not decidable whether or not a value of B will always return a number on every input. Instead, following [Findler and Felleisen 2002], gradual type casts *wrap* the function with a cast on its outputs and if at any point it returns something that is not a number, a type error is raised. So if $v : B$ is cast to A and back, we cannot expect to always get an equivalent value back, but the result should *error more*—that is, either the cast to A raises an error, or we get back a new value $v' : B$ that has the same behavior as v except it sometimes raises a type error. We formalize this as *observational error approximation* and write the ordering $t \sqsubseteq t'$ as "t errors more than t'". We then use this to formalize the *projection* property: $\langle B \Leftarrow A\rangle\langle A \Leftarrow B\rangle s \sqsubseteq s$.

Notice how the justification for the projection property uses the same intuition as graduality: that casts should only be doing *checking* and not completely changing a program's behavior. This is the key to why embedding-projection pairs help to formulate and prove graduality: we view graduality as the natural extension of the projection property from a property of casts to a property of arbitrary gradually typed programs.

This gives us nice properties of some casts, but what do we know about casts that are *not* upcasts or downcasts? In traditional formulations, gradual typing includes casts $\langle B \Leftarrow A\rangle$ between types that are *shallowly compatible*—i.e, that are not guaranteed to fail. For instance, we can cast a pair where the left side is known to be a number $\mathbb{N} \times ?$ to a type where the right side is known to be a number $? \times \mathbb{N}$ with casts succeeding on values where both sides are numbers. The resulting cast $\langle ? \times \mathbb{N} \Leftarrow \mathbb{N} \times ?\rangle$ is neither an upcast nor a downcast. We argue that the formulation based on these

[2]It is not literally true that every gradual language uses this presentation of casts from cast calculi, but in order for a language to be gradually typed, some means of casting between types must be available, such as embedding dynamic code in statically typed code, or type annotations. We argue that the properties of casts we identify here should apply to those presentations as well.

"general" casts is ill behaved from a meta-theoretic perspective: you are quite limited in your ability to break casts for larger types into casts for smaller types. Most notably, the *composition* of two general casts is very rarely the same as the direct cast. For instance, casting from $\mathbb{N}$ to $? \to ?$ and back to $\mathbb{N}$ always errors, but obviously the direct cast $\langle \mathbb{N} \Leftarrow \mathbb{N} \rangle$ is the identity. We show that upcast and downcasts on the other hand satisfy a *decomposition* theorem: if $A_1 \sqsubseteq A_2 \sqsubseteq A_3$, then the upcast from A_1 to A_3 factors through A_2 and similarly for the downcast.

Furthermore, if we disregard *performance* of the casts, and only care about the observational behavior, we show that any "general" cast is the composition of an upcast followed by a downcast.[3] For instance, our cast from before $\langle ? \times \mathbb{N} \Leftarrow \mathbb{N} \times ? \rangle$ is observationally equivalent to the composition of first *up*casting to a pair where both sides are dynamically typed $? \times ?$ and then *down*casting: $\langle ? \times \mathbb{N} \Leftarrow ? \times ? \rangle \langle ? \times ? \Leftarrow \mathbb{N} \times ? \rangle$. We show that *all* the casts in a standard gradually typed language exhibit this factorization, which means that for the purposes of formulating and proving graduality, we need only discuss upcasts and downcasts. For implementation, it is more convenient to have a primitive notion of coercion/direct cast to eliminate/collapse casts [Herman et al. 2010; Siek and Wadler 2010], but we argue that the correctness of such an implementation should be justified by a simulation relation with a simpler semantics, meaning the implementation would inherit a proof of graduality from the simpler semantics as well.

To prove these equivalence and approximation results, we develop a novel step-indexed logical relation that is sound for observational error approximation. We also develop high-level reasoning principles from the relation so that our main lemmas do not involve any manual step-manipulation.

Finally, based on our semantic interpretation of type dynamism as embedding-projection pairs, we provide a refined analysis of the proof theory of type dynamism as a *syntax for building ep pairs*. We give a semantics for these proof terms analogous to the coercion interpretation of subtyping derivations. Similar to subtyping, we prove a *coherence* theorem which gives, as a corollary, our decomposition theorem for upcasts and downcasts.

Graduality. In Siek et al. [2015], they prove the (dynamic) gradual guarantee by an operational simulation argument whose details are quite tied to the specific cast calculus used. Using the ep pairs, we provide a more semantic formulation and proof of graduality. First, we use our analysis of type dynamism as denoting ep pairs to define graduality as a kind of observational error approximation *up to upcast/downcast*, building on the axiomatic semantics of graduality in New and Licata [2018]. We then prove the graduality theorem using our logical relation for error approximation. Notably, the decomposition theorem for ep pairs leads to a clean, uniform proof of the cast case of graduality.

Overview of Technical Development and Contributions. In this paper, we show how to prove graduality for a standard gradually typed cast calculus by translating it into a simple typed language with recursive types and errors. Specifically, our development proceeds as follows:

(1) We present a standard gradually typed cast calculus (λ_G) and its operational semantics, using "general" casts (§2).
(2) We present a simple typed language with recursive types and a type error ($\lambda_{T,\mho}$), into which we translate the cast calculus. Casts in λ_G are translated to contracts implemented in the typed language (§3).
(3) We develop a novel step-indexed logical relation that is sound for our notion of observational error approximation (§4). We prove transitivity of the logical relation and other high-level reasoning principles so that our main lemmas for ep-pairs and graduality do not involve any manual step-manipulation.

[3]Note that this is not the same as the factorization of casts known as "threesomes", see §7 for a comparison.

$$
\begin{array}{llcl}
\text{Types} & \mathsf{A, B} & ::= & \mathsf{?} \mid \mathsf{1} \mid \mathsf{A \times B} \mid \mathsf{A + B} \mid \mathsf{A \to B} \\
\text{Tags} & \mathsf{G} & ::= & \mathsf{1} \mid \mathsf{? \times ?} \mid \mathsf{? + ?} \mid \mathsf{? \to ?} \\
\text{Terms} & \mathsf{t, s} & ::= & \mho \mid \mathsf{x} \mid \langle \mathsf{B} \Leftarrow \mathsf{A} \rangle \mathsf{t} \mid \langle\rangle \mid \langle \mathsf{t_1, t_2} \rangle \mid \mathsf{let}\, \langle \mathsf{x, y} \rangle = \mathsf{t}\, \mathsf{in}\, \mathsf{s} \\
 & & \mid & \mathsf{inj}\, \mathsf{t} \mid \mathsf{inj}'\, \mathsf{t} \mid \mathsf{case}\, \mathsf{t}\, \mathsf{of}\, \ \mathsf{inj}\, \mathsf{x_1}.\, \mathsf{t_1} \mid \mathsf{inj}'\, \mathsf{x_2}.\, \mathsf{t_2} \mid \lambda(\mathsf{x} : \mathsf{A}).\, \mathsf{t} \mid \mathsf{t\, s} \\
\text{Values} & \mathsf{v} & ::= & \langle ? \Leftarrow \mathsf{G} \rangle \mathsf{v} \mid \langle\rangle \mid \langle \mathsf{v_1, v_2} \rangle \mid \mathsf{inj}\, \mathsf{v} \mid \mathsf{inj}'\, \mathsf{v} \mid \lambda(\mathsf{x} : \mathsf{A}).\, \mathsf{t} \\
\text{Evaluation Contexts} & \mathsf{E} & ::= & [\cdot] \mid \langle \mathsf{B} \Leftarrow \mathsf{A} \rangle \mathsf{E} \mid \langle \mathsf{E, s} \rangle \mid \langle \mathsf{v, E} \rangle \mid \mathsf{let}\, \langle \mathsf{x, y} \rangle = \mathsf{E}\, \mathsf{in}\, \mathsf{s} \\
 & & \mid & \mathsf{inj}\, \mathsf{E} \mid \mathsf{inj}'\, \mathsf{E} \mid \mathsf{case}\, \mathsf{E}\, \mathsf{of}\, \ \mathsf{inj}\, \mathsf{x_1}.\, \mathsf{t_1} \mid \mathsf{inj}'\, \mathsf{x_2}.\, \mathsf{t_2} \mid \mathsf{E\, s} \mid \mathsf{v\, E} \\
\text{Environments} & \Gamma & ::= & \cdot \mid \Gamma, \mathsf{x} : \mathsf{A} \\
\text{Substitutions} & \gamma & ::= & \cdot \mid \gamma, \mathsf{v}/\mathsf{x}
\end{array}
$$

Fig. 1. λ_G Syntax

$$\boxed{\Gamma \vdash \mathsf{t} : \mathsf{A}}$$

$$
\frac{}{\Gamma \vdash \mho : \mathsf{A}} \qquad
\frac{}{\Gamma, \mathsf{x} : \mathsf{A}, \Gamma' \vdash \mathsf{x} : \mathsf{A}} \qquad
\frac{\Gamma \vdash \mathsf{t} : \mathsf{A}}{\Gamma \vdash \langle \mathsf{B} \Leftarrow \mathsf{A} \rangle \mathsf{t} : \mathsf{B}} \qquad
\frac{}{\Gamma \vdash \langle\rangle : \mathsf{1}} \qquad
\frac{\Gamma \vdash \mathsf{t_1} : \mathsf{A_1} \qquad \Gamma \vdash \mathsf{t_2} : \mathsf{A_2}}{\Gamma \vdash \langle \mathsf{t_1, t_2} \rangle : \mathsf{A_1 \times A_2}}
$$

$$
\frac{\Gamma \vdash \mathsf{t} : \mathsf{A_1 \times A_2} \qquad \Gamma, \mathsf{x} : \mathsf{A_1}, \mathsf{y} : \mathsf{A_2} \vdash \mathsf{s} : \mathsf{B}}{\Gamma \vdash \mathsf{let}\, \langle \mathsf{x, y} \rangle = \mathsf{t}\, \mathsf{in}\, \mathsf{s} : \mathsf{B}} \qquad
\frac{\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{t} : \mathsf{B}}{\Gamma \vdash \lambda(\mathsf{x} : \mathsf{A}).\, \mathsf{t} : \mathsf{A \to B}} \qquad
\frac{\Gamma \vdash \mathsf{t} : \mathsf{A \to B} \qquad \Gamma \vdash \mathsf{s} : \mathsf{A}}{\Gamma \vdash \mathsf{t\, s} : \mathsf{B}}
$$

Fig. 2. λ_G Typing Rules (excerpt)

(4) We present a novel analysis of type dynamism as a coherent syntax for ep pairs and show that all of the casts of the gradual language can be factorized as an upcast followed by a downcast (§5).
(5) We give a semantic formulation of graduality and then prove it using our error-approximation logical relation and ep pairs (§6).

Proofs and definitions elided from this paper are presented in full in the extended version of the paper [New and Ahmed 2018].

2 GRADUAL CAST CALCULUS

Our starting point is a fairly typical gradual cast calculus, called λ_G, in the style of Wadler and Findler [2009] and Siek et al. [2015]. A cast calculus is usually the target of an elaboration pass from a gradually typed surface language. The gradually typed surface language makes mixing static and dynamic code seamless, for instance a typed function on numbers $f : \mathbb{N} \to \mathbb{N}$ can be applied to a dynamically typed value $x : ?$ and the result is well typed $f(x) : \mathbb{N}$. Since x is not known to be a number, at runtime a dynamic check is performed: if x is a number, f is run with its value and otherwise a dynamic type error is raised. In the surface language, this checking behavior takes place at every elimination form: pattern matching, referencing a field, etc. The cast calculus makes the dynamic type checking separate from the elimination forms using explicit cast forms. If $\mathsf{t} : \mathsf{A}$ in the cast calculus, then we can cast it to another type B using the cast form $\langle \mathsf{B} \Leftarrow \mathsf{A} \rangle \mathsf{t}$. This means we can use the ordinary typed reduction rules for elimination forms, and all the details of checking are isolated to the cast reductions. We choose to use a cast calculus, rather than a gradual surface language, since we are chiefly concerned with the semantics of the language, rather than gradual type checking.

We present the syntax of λ_G (pronounced "lambda gee" and typeset in blue sans-serif font) in Figure 1, and most ofthe typing rules in Figure 2. The language is call-by-value and includes standard type formers, namely, the unit type $\mathsf{1}$, product type $\times$, sum type $+$, and function type $\to$, with standard typing rules. The language also includes some features specific to gradual typing: a dynamic type $\mathsf{?}$, a dynamic type error $\mho$ and casts $\langle \mathsf{B} \Leftarrow \mathsf{A} \rangle \mathsf{t}$. Following previous work, the interface for the dynamic type $\mathsf{?}$ is given by the casts themselves, and not distinct introduction and

$\boxed{\lfloor A \rfloor \overset{\text{def}}{=} G}$ where $A \neq ?$

$$\begin{aligned}\lfloor 1 \rfloor &\overset{\text{def}}{=} 1\\ \lfloor A \times B \rfloor &\overset{\text{def}}{=} ? \times ?\\ \lfloor A + B \rfloor &\overset{\text{def}}{=} ? + ?\\ \lfloor A \to B \rfloor &\overset{\text{def}}{=} ? \to ?\end{aligned}$$

Fig. 3. λ_G: Tag of a (non-dynamic) Type

$$E[\mathsf{case}\,(\mathsf{inj}\,v)\,\mathsf{of}\;\;\mathsf{inj}\,x.\,t \mid \mathsf{inj}'\,x'.\,t'] \mapsto E[t[v/x]] \qquad E[\mathsf{case}\,(\mathsf{inj}'\,v)\,\mathsf{of}\;\;\mathsf{inj}\,x.\,t \mid \mathsf{inj}'\,x'.\,t'] \mapsto E[t'[v/x']]$$

$$E[\mathsf{let}\,\langle x_1, x_2\rangle = \langle v_1, v_2\rangle\,\mathsf{in}\,t] \mapsto E[t[v_1/x_1, v_2/x_2]] \qquad E[(\lambda x.\,t)\;v] \mapsto E[t[v/x]] \qquad E[\mho] \mapsto \mho$$

$$E[\langle ? \Leftarrow ?\rangle v] \mapsto E[v]\ \textsc{DynDyn} \qquad \frac{A \neq ? \qquad \lfloor A \rfloor \neq A}{E[\langle ? \Leftarrow A\rangle v] \mapsto E[\langle ? \Leftarrow \lfloor A \rfloor\rangle(\langle \lfloor A \rfloor \Leftarrow A\rangle v)]}\ \textsc{TagUp}$$

$$\frac{A \neq ? \qquad \lfloor A \rfloor \neq A}{E[\langle A \Leftarrow ?\rangle v] \mapsto E[\langle A \Leftarrow \lfloor A \rfloor\rangle\langle \lfloor A \rfloor \Leftarrow ?\rangle v]}\ \textsc{TagDn} \qquad E[\langle G \Leftarrow ?\rangle\langle ? \Leftarrow G\rangle v] \mapsto E[v]\ \textsc{TagMatch}$$

$$\frac{G \neq G'}{E[\langle G \Leftarrow ?\rangle\langle ? \Leftarrow G'\rangle v] \mapsto \mho}\ \textsc{TagMismatch} \qquad \frac{A, B \neq ? \qquad \lfloor A \rfloor \neq \lfloor B \rfloor}{E[\langle B \Leftarrow A\rangle v] \mapsto \mho}\ \textsc{TagMismatch'}$$

$$E[\langle A_2 \times B_2 \Leftarrow A_1 \times B_1\rangle\langle v, v'\rangle] \mapsto E[\langle\langle A_2 \Leftarrow A_1\rangle v, \langle B_2 \Leftarrow B_1\rangle v'\rangle]\ \textsc{Pair}$$

$$E[\langle A_2 + B_2 \Leftarrow A_1 + B_1\rangle \mathsf{inj}\,v] \mapsto E[\langle A_2 \Leftarrow A_1\rangle v]\ \textsc{Sum}$$

$$E[\langle A_2 + B_2 \Leftarrow A_1 + B_1\rangle \mathsf{inj}'\,v] \mapsto E[\langle B_2 \Leftarrow B_1\rangle v]\ \textsc{Sum'}$$

$$E[\langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle v] \mapsto E[\lambda(x : A_2).\,\langle B_2 \Leftarrow B_1\rangle(v\;(\langle A_1 \Leftarrow A_2\rangle x))]\ \textsc{Fun}$$

Fig. 4. λ_G Operational Semantics: non-casts (top) and casts (bottom)

elimination forms. The values of the dynamic type are of the form $\langle ? \Leftarrow G\rangle v$ where G ranges over *tag types*, defined in Figure 1. The tag types are so called because they represent the "tags" used to distinguish between the basic sorts of dynamically typed values. Every type except $?$ has an "underlying" tag type we write as $\lfloor A \rfloor$ and define in Figure 3. These tag types are the cases of the dynamic type $?$ seen as a sum type, which is how we model it in §3.2. For any two types A, B, we can form the cast $\langle B \Leftarrow A\rangle$ which at runtime will attempt to coerce a term $v : A$ into a valid term of type B. If the value cannot sensibly be interpreted as a value in B, the cast *fails* and reduces to the dynamic type error $\mho$. The type error is like an uncatchable exception, modeling the fact that the program crashes with an error message when a dynamic type error is encountered. In this paper we consider all type errors to be equivalent. The calculus is based on that of Wadler and Findler [2009], but does not have blame and removes the restriction that types must be compatible in order to define a cast.

Figure 4 presents the operational semantics of the gradual language in the style of Felleisen and Hieb [1992], using *evaluation contexts* E to specify a left-to-right, call-by-value evaluation order. The top of the figure shows the reductions *not* involving casts. This includes the standard reductions for pairs, sums, and functions using the obvious notion of substitution $t[\gamma]$, in addition to a reduction $E[\mho] \mapsto \mho$ to propagate a dynamic type error to the top level.

Types	A, B	$::=$	$\mu\alpha.A \mid \alpha \mid 1 \mid A \times B \mid A + B \mid A \to B$
Terms	t, s	$::=$	$\mho \mid x \mid \mathrm{let}\, x = t \,\mathrm{in}\, s \mid \mathrm{roll}_A\, t \mid \mathrm{unroll}\, t \mid \langle\rangle \mid \langle t, s\rangle$
		$\mid$	$\mathrm{let}\, \langle x, y\rangle = t \,\mathrm{in}\, s \mid \mathrm{inj}\, t \mid \mathrm{inj}'\, t$
		$\mid$	$\mathrm{case}\, t \,\mathrm{of}\ \ \mathrm{inj}\, x_1 . t_1 \mid \mathrm{inj}'\, x_2 . t_2 \mid \lambda(x : A).t \mid t\ s$
Values	v	$::=$	$x \mid \mathrm{roll}_A\, v \mid \langle\rangle \mid \langle v, v\rangle \mid \mathrm{inj}\, v \mid \mathrm{inj}'\, v \mid \lambda(x : A).t$
Evaluation Contexts	E	$::=$	$[\cdot] \mid \mathrm{let}\, x = E \,\mathrm{in}\, s \mid \mathrm{roll}_A\, E \mid \mathrm{unroll}\, E \mid \langle E, t\rangle \mid E\ s \mid v\ E \mid \langle v, E\rangle$
		$\mid$	$\mathrm{let}\, \langle x, y\rangle = E \,\mathrm{in}\, s \mid \mathrm{inj}\, E \mid \mathrm{inj}'\, E \mid \mathrm{case}\, E \,\mathrm{of}\ \ \mathrm{inj}\, x_1 . t_1 \mid \mathrm{inj}'\, x_2 . t_2$
Environments	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Substitutions	γ	$::=$	$\cdot \mid \gamma, v/x$

Fig. 5. $\lambda_{T,\mho}$ Syntax

More importantly, the bottom of the figure shows the reductions of *casts*, specifying the dynamic type checking necessary for gradual typing. First (DynDyn), casting from dynamic to itself is the identity. For any type A that is not a tag type (checked by $\lfloor A \rfloor \neq A$) or the dynamic type, casting to the dynamic type first casts to its underlying tag type $\lfloor A \rfloor$ and then tags it at that type (TagUp). Similarly, casting down from the dynamic type first casts to the underlying tag type (TagDn). The next two rules are the primitive reductions for tags: if you project at the correct tag type, you get the underlying value out (TagMatch) and otherwise a dynamic type error is raised (TagMismatch). Similarly, the next rule (TagMismatch') says that if two types are incompatible in that they have distinct tag types and neither is dynamic, then the cast errors. The next three (Fun, Pair, Sum) are the standard "wrapping" implementations of contracts/casts [Findler and Felleisen 2002], also familiar from subtyping. For the function cast $\langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle$, note that while the output type is the same direction $\langle B_2 \Leftarrow B_1\rangle$, the input cast is flipped: $\langle A_1 \Leftarrow A_2\rangle$.

We note that this standard operational semantics is quite complex for such a *small* language. In particular, it is more complicated than the operational semantics of typed and dynamically typed languages of similar size. Typed languages have reductions for each elimination form and dynamically typed languages add only the possibility of type error to those reductions. Here on the other hand, the semantics is *not* modular in the same way: there are five rules involving the dynamic type and four of them involve comparing arbitrary types.

For these reasons, we find the cast calculus presentation inconvenient for semantic analysis, and we choose not to develop our theory of graduality or even prove type safety *directly* for this language. Instead, we will *translate* the cast calculus into a typed language where the casts are translated to functions implemented in the language, i.e. contracts [Findler and Felleisen 2002]. This has the advantage of reducing the size of the language, making "language-level" theorems like type safety and soundness of a logical relation easier to prove. Finally, note that our central theorems are still about the gradual language, but we will prove them by lifting results about their translations using an *adequacy* theorem (theorem 3.7).

3 TRANSLATING GRADUAL TYPING

We now translate our cast calculus into a simpler, non-gradual typed language with errors. We then prove an adequacy theorem that enables us to prove theorems about gradual programs by reasoning about their translations.

3.1 Typed Language with Errors

The typed language we will translate into is $\lambda_{T,\mho}$ (pronounced "lambda tee error" and typeset in **bold red serif font**), a call-by-value typed lambda calculus with iso-recursive types and an uncatchable error. Figure 5 shows the syntax of the language. Figure 6 shows some of the typing rules; the rest are completely standard.

$$\boxed{\Gamma \vdash t : A}$$

$$\frac{}{\Gamma \vdash \mho : A} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A[\mu\alpha.A/\alpha]}{\Gamma \vdash \mathbf{roll}_{\mu\alpha.A}\, t : \mu\alpha.A} \qquad \frac{\Gamma \vdash t : \mu\alpha.A}{\Gamma \vdash \mathbf{unroll}\, t : A[\mu\alpha.A/\alpha]}$$

Fig. 6. $\lambda_{T,\mho}$ Typing Rules (excerpt)

$$E[\mho] \mapsto^0 \mho \qquad E[\mathbf{let}\, x = v\, \mathbf{in}\, s] \mapsto^0 E[s[v/x]] \qquad E[\mathbf{unroll}\,(\mathbf{roll}_A\, v)] \mapsto^1 E[v]$$

$$E[\mathbf{let}\,\langle x_1, x_2\rangle = \langle v_1, v_2\rangle\, \mathbf{in}\, t] \mapsto^0 E[t[v_1/x_1, v_2/x_2]] \qquad E[(\lambda(x:A).t)\, v] \mapsto^0 E[t[v/x]]$$

$$E[\mathbf{case}\,(\mathbf{inj}\, v)\, \mathbf{of}\;\; \mathbf{inj}\, x.t \mid \mathbf{inj}'\, x'.t'] \mapsto^0 E[t[v/x]]$$

$$E[\mathbf{case}\,(\mathbf{inj}'\, v)\, \mathbf{of}\;\; \mathbf{inj}\, x.t \mid \mathbf{inj}'\, x'.t'] \mapsto^0 E[t'[v/x']]$$

$$\frac{}{t \Mapsto^0 t} \qquad \frac{t \mapsto^i t' \qquad t' \Mapsto^j t''}{t \Mapsto^{i+j} t''}$$

Fig. 7. $\lambda_{T,\mho}$ Operational Semantics

$$\begin{aligned}
[\![?]\!] &\overset{\text{def}}{=} \mu\alpha.1 + (\alpha \times \alpha) + (\alpha + \alpha) + (\alpha \to \alpha)\\
[\![1]\!] &\overset{\text{def}}{=} 1\\
[\![A \times B]\!] &\overset{\text{def}}{=} [\![A]\!] \times [\![B]\!]\\
[\![A + B]\!] &\overset{\text{def}}{=} [\![A]\!] + [\![B]\!]\\
[\![A \to B]\!] &\overset{\text{def}}{=} [\![A]\!] \to [\![B]\!]
\end{aligned}$$

Fig. 8. Type Translation

The types of the language are similar to the cast calculus: they include the standard type formers of products, sums, and functions. Rather than the specific dynamic type, we include the more general, but standard, iso-recursive type $\mu\alpha.A$, which is isomorphic to the unfolding $A[\mu\alpha.A/\alpha]$ by the terms $\mathbf{roll}_{\mu\alpha.A}\,\cdot$ and $\mathbf{unroll}\,\cdot$. As in the source language we have an uncatchable error $\mho$.

Figure 7 presents the operational semantics of the language. For the purposes of later defining a step-indexed logical relation, we assign a weight to each small step of the operational semantics that is 1 for unrolling a value of recursive type and 0 for other reductions. We then define a "quantitative" reflexive, transitive closure of the small-step relation $t \Mapsto^i t'$ that adds the weights of its constituent small steps. When the number of steps is irrelevant, we just use $\mapsto$ and $\Mapsto$. We can then establish some simple facts about this operational semantics.

Lemma 3.1 (Subject Reduction). *If* $\cdot \vdash t : A$ *and* $t \Mapsto t'$ *then* $\cdot \vdash t' : A$.

Lemma 3.2 (Progress). *If* $\cdot \vdash t : A$ *and* t *is not a value or* $\mho$*, then there exists* t' *with* $t \mapsto t'$.

Lemma 3.3 (Determinism). *If* $t \mapsto s$ *and* $t \mapsto s'$*, then* $s = s'$.

3.2 Translating Gradual Typing

Next we translate the cast calculus into our typed language, and prove that the cast calculus semantics is in a simulation relation with the typed language. Since the two languages share so much of their syntax, most of the translation is a simple "color change", only the parts that are truly components of gradual typing need much translation.

Our translation is type preserving, so we first define a *type translation* in Figure 8. The dynamic type is interpreted as a recursive sum of the translations of the tag types of the gradual language.

$\boxed{[\![t]\!]}$ where if $x_1 : A_1, \dots, x_n : A_n \vdash t : A$ then $\boldsymbol{x}_1 : [\![A_1]\!], \dots, \boldsymbol{x}_n : [\![A_n]\!] \vdash [\![t]\!] : [\![A]\!]$

$$
\begin{aligned}
[\![x]\!] &\overset{\text{def}}{=} \boldsymbol{x} \\
[\![\langle B \Leftarrow A\rangle t]\!] &\overset{\text{def}}{=} E_{\langle B \Leftarrow A\rangle}[[\![t]\!]] \\
[\![\langle\rangle]\!] &\overset{\text{def}}{=} \langle\rangle \\
[\![\langle t_1, t_2\rangle]\!] &\overset{\text{def}}{=} \langle [\![t_1]\!], [\![t_2]\!]\rangle \\
[\![\mathrm{let}\ \langle x, y\rangle = t\ \mathrm{in}\ s]\!] &\overset{\text{def}}{=} \mathbf{let}\ \langle \boldsymbol{x}, \boldsymbol{y}\rangle = [\![t]\!]\ \mathbf{in}\ [\![s]\!] \\
[\![\mathrm{inj}\ t]\!] &\overset{\text{def}}{=} \mathbf{inj}\ [\![t]\!] \\
[\![\mathrm{inj}'\ t]\!] &\overset{\text{def}}{=} \mathbf{inj}'\ [\![t]\!] \\
[\![\mathrm{case}\ t\ \mathrm{of}\ \ \mathrm{inj}\ x. s \mid \mathrm{inj}'\ x'. s']\!] &\overset{\text{def}}{=} \mathbf{case}\ [\![t]\!]\ \mathbf{of}\ \ \mathbf{inj}\ \boldsymbol{x}. [\![s]\!] \mid \mathbf{inj}'\ \boldsymbol{x}'. [\![s']\!] \\
[\![\lambda(x : A). t]\!] &\overset{\text{def}}{=} \lambda(\boldsymbol{x} : [\![A]\!]). [\![t]\!] \\
[\![t\ s]\!] &\overset{\text{def}}{=} [\![t]\!]\ [\![s]\!]
\end{aligned}
$$

Fig. 9. Term Translation

$\boxed{E_{\langle B \Leftarrow A\rangle}}$ where $\boldsymbol{x} : [\![A]\!] \vdash E_{\langle B \Leftarrow A\rangle}[\boldsymbol{x}] : [\![B]\!]$

$$
\begin{aligned}
E_{\langle ? \Leftarrow ?\rangle} &\overset{\text{def}}{=} [\cdot] \\
E_{\langle A_2 \times B_2 \Leftarrow A_1 \times B_1\rangle} &\overset{\text{def}}{=} E_{\langle A_2 \Leftarrow A_1\rangle} \times E_{\langle B_2 \Leftarrow B_1\rangle} \\
E_{\langle A_2 + B_2 \Leftarrow A_1 + B_1\rangle} &\overset{\text{def}}{=} E_{\langle A_2 \Leftarrow A_1\rangle} + E_{\langle B_2 \Leftarrow B_1\rangle} \\
E_{\langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle} &\overset{\text{def}}{=} E_{\langle A_1 \Leftarrow A_2\rangle} \to E_{\langle B_2 \Leftarrow B_1\rangle} \\
E_{\langle ? \Leftarrow G\rangle} &\overset{\text{def}}{=} \mathbf{roll}_{[\![?]\!]}\ \mathbf{inj}_G\ [\cdot] \\
E_{\langle G \Leftarrow ?\rangle} &\overset{\text{def}}{=} \mathbf{case}\ (\mathbf{unroll}\ [\cdot])\ \mathbf{of}\ \mathbf{inj}_G\ \boldsymbol{x}. \boldsymbol{x} \mid \mathbf{else}. \mho \\
E_{\langle ? \Leftarrow A\rangle} &\overset{\text{def}}{=} E_{\langle ? \Leftarrow \lfloor A\rfloor\rangle}[E_{\langle \lfloor A\rfloor \Leftarrow A\rangle}[\cdot]] \qquad \text{if } A \neq ?, \lfloor A\rfloor \\
E_{\langle A \Leftarrow ?\rangle} &\overset{\text{def}}{=} E_{\langle A \Leftarrow \lfloor A\rfloor\rangle}[E_{\langle \lfloor A\rfloor \Leftarrow ?\rangle}[\cdot]] \qquad \text{if } A \neq ?, \lfloor A\rfloor \\
E_{\langle B \Leftarrow A\rangle} &\overset{\text{def}}{=} \mathbf{let}\ \boldsymbol{x} = [\cdot]\ \mathbf{in}\ \mho \qquad \text{if } A, B \neq ? \text{ and } \lfloor A\rfloor \neq \lfloor B\rfloor
\end{aligned}
$$

Fig. 10. Direct Cast Translation

$$
\begin{aligned}
E \times E' &\overset{\text{def}}{=} \mathbf{let}\ \langle \boldsymbol{x}, \boldsymbol{x}'\rangle = [\cdot]\ \mathbf{in}\ \langle E[\boldsymbol{x}], E'[\boldsymbol{x}']\rangle \\
E + E' &\overset{\text{def}}{=} \mathbf{case}\ [\cdot]\ \mathbf{of}\ \ \mathbf{inj}\ \boldsymbol{x}. E[\boldsymbol{x}] \mid \mathbf{inj}'\ \boldsymbol{x}'. E'[\boldsymbol{x}'] \\
E \to E' &\overset{\text{def}}{=} \mathbf{let}\ \boldsymbol{x}_f = [\cdot]\ \mathbf{in}\ \lambda \boldsymbol{x}_a. E'[\boldsymbol{x}_f\ (E[\boldsymbol{x}_a])]
\end{aligned}
$$

Fig. 11. Functorial Action of Type Connectives

The unit, pair, sum and function types are all interpreted as the corresponding connectives in the typed language.

Next, we define the translation of terms in Figure 9, which is type preserving in that if $x_1 : A_1, \dots, x_n : A_n \vdash t : A$ then $\boldsymbol{x}_1 : [\![A_1]\!], \dots, \boldsymbol{x}_n : [\![A_n]\!] \vdash [\![t]\!] : [\![A]\!]$. Again, most of the translation is just a change of hue. The most important rule of the term translation is that of casts. A cast $\langle B \Leftarrow A\rangle$ is translated to an *evaluation context* $E_{\langle B \Leftarrow A\rangle}$ of the appropriate type, which are defined in Figure 10. Each case of the definition corresponds to one or more rules of the operational semantics. The product, sum, and function rules use the definitions of functorial actions of their types from Figure 11. We separate them because we will use the functoriality property in several definitions, theorems, and proofs later.

3.3 Operational Properties

Next, we consider the relationship between the operational semantics of the two languages and how to lift properties of the typed language to the gradual language. We want to view the translation of the cast calculus into the typed language as *definitional*, and in that regard view the operational semantics of the source language as being based on the typed language. We capture this relationship in the following *forward* simulation theorem, which says that any reduction in the cast calculus corresponds to (and is *justified by*) multiple steps in the target:

Lemma 3.4 (Translation Preserves Values, Evaluation Contexts).

(1) *For any value* v, $[\![\mathsf{v}]\!]$ *is a value.*
(2) *For any evaluation context* E, $[\![\mathsf{E}]\!]$ *is an evaluation context.*

Lemma 3.5 (Simulation of Operational Semantics).
If $\mathsf{t} \mapsto \mathsf{t}'$ *then there exists* $\boldsymbol{s}$ *with* $[\![\mathsf{t}]\!] \mapsto \boldsymbol{s}$ *and* $\boldsymbol{s} \Mapsto [\![\mathsf{t}']\!]$.

To lift theorems for the gradual language from the typed language, we need to establish an *adequacy* theorem, which says that the operational behavior of a translated term determines the behavior of the original source term. To do this we use the following backward simulation theorem.

Lemma 3.6 (Backward Simulation).

(1) *If* $[\![\mathsf{t}]\!]$ *is a value,* $\mathsf{t} \mapsto^{*} \mathsf{v}$ *for some* v *with* $[\![\mathsf{t}]\!] = [\![\mathsf{v}]\!]$.
(2) *If* $[\![\mathsf{t}]\!] = \mho$, *then* $\mathsf{t} \mapsto^{*} \mho$.
(3) *If* $[\![\mathsf{t}]\!] \mapsto \boldsymbol{s}$ *then there exists* s' *with* $\mathsf{t} \mapsto \mathsf{s}'$ *and* $\boldsymbol{s} \Mapsto [\![\mathsf{s}']\!]$.

Theorem 3.7 (Adequacy).

(1) $[\![\mathsf{t}]\!] \Mapsto \boldsymbol{v}$ *if and only if* $\mathsf{t} \mapsto^{*} \mathsf{v}$ *with* $[\![\mathsf{v}]\!] = \boldsymbol{v}$.
(2) $[\![\mathsf{t}]\!] \Mapsto \mho$ *if and only if* $\mathsf{t} \mapsto^{*} \mho$.
(3) $[\![\mathsf{t}]\!]$ *diverges if and only if* t *diverges*

While this has reduced the number of primitives of the language, reasoning about the behavior of the translated casts isn't any simpler than the original operational semantics since they have the same behavior. For simpler reasoning about cast behavior, we will move further away from a direct simulation of the source operational semantics, to a second semantics based on ep pairs that is observationally equivalent but also conceptually simpler and helps prove the gradual guarantee. However, in order to prove that the second semantics is equivalent, we first need to develop a usable theory of observational equivalence and approximation.

4 A LOGICAL RELATION FOR ERROR APPROXIMATION

Next, we define observational equivalence and error approximation of programs in the gradual and typed languages, the two properties with which we formulate embedding-projection pairs. To facilitate proofs of error approximation, we develop a novel step-indexed logical relation. Since our notion of approximation is non-standard, the use of step-indexing in our logical relation is inconvenient to use directly. So, on top of the "implementation" of the logical relation as a step-indexed relation, we prove many high-level lemmas so that all proofs in the next sections are performed relative to these lemmas, and none manipulate step indices directly.

4.1 Observational Equivalence and Approximation

A suitable notion of equivalence for programs is *observational equivalence.* We say t is observationally equivalent to t' if replacing one with the other in the context of a larger program produces the same result (termination, error, or divergence). We formalize this saying a program *context* C is a term with a single hole $[\cdot]$. A context is typed $\Gamma' \vdash C[\Gamma \vdash \cdot : A] : A'$ when for any term $\Gamma \vdash t : A$, replacing the hole with t results in a well-typed $\Gamma' \vdash C[t] : A'$

While this notion of observational equivalence is entirely standard, the notion of *approximation* we use—which we call error approximation—is not the standard notion of observational approximation. Usually, we would say t observationally approximates t' if, when placing them into the same context C, either $C[t]$ diverges or they both terminate or both error. We call this form of approximation *divergence approximation*. However, for *gradual typing* we are not particularly interested in when one program diverges more than another, but rather when it produces more *type errors*. We might be tempted to conflate the two, but their behavior is quite distinct! We can never truly know if a black-box program will continue indefinitely: it would frustrate any programmer to use a language that runs forever when accidentally using a function as a number. The reader should keep this difference in mind when seeing how our logical relation differs form the standard treatment. In the rest of this paper, when discussing the two together we will clearly distinguish between divergence and error approximation, but when there is no qualifier, approximation is meant as *error approximation*.

Definition 4.1 (Gradual Observational Equivalence, Error Approximation). For any well typed terms $\Gamma \vdash t, t' : A$,

(1) Define $\Gamma \vDash t \approx^{\text{obs}} t' : A$, pronounced "$t$ is observationally equivalent to t'" to hold when for any $\cdot \vdash C[\Gamma \vdash \cdot : A] : B$, either $C[t]$ and $C[t']$ both reduce to a value, both reduce to an error, or both diverge.
(2) Define $\Gamma \vDash t \sqsubseteq^{\text{obs}} t' : A$, pronounced "$t$ observationally (error) approximates t'" to hold when for any $\cdot \vdash C[\Gamma \vdash \cdot : A] : B$, either $C[t]$ reduces to $\mho$ or both $C[t]$ and $C[t']$ reduce to a value or both diverge.

As with divergence approximation, we can prove two programs are observationally equivalent by showing each error approximates the other.

Lemma 4.2 (Equivalence is Approximation Both Ways). *$\Gamma \vDash t_1 \approx^{obs} t_2 : A$ if and only if both $\Gamma \vDash t_1 \sqsubseteq^{obs} t_2 : A$ and $\Gamma \vDash t_2 \sqsubseteq^{obs} t_1 : A$.*

We define typed observational equivalence $\boldsymbol{\Gamma} \vDash \boldsymbol{t} \approx^{\text{obs}} \boldsymbol{t'} : \boldsymbol{A}$ and observational error approximation $\boldsymbol{\Gamma} \vDash \boldsymbol{t} \sqsubseteq^{\text{obs}} \boldsymbol{t'} : \boldsymbol{A}$ with the exact same definition as for the gradual language above, but in **red** instead of blue. We rarely work with the gradual language directly, instead we prove approximation results for their translations. This is justified by the following lemma, a consequence of our *adequacy* result (theorem 3.7).

Lemma 4.3 (Typed Observational Approximation implies Gradual Observational Approximation). *If $[\![\Gamma]\!] \vDash [\![t_1]\!] \sqsubseteq^{obs} [\![t_2]\!] : [\![A]\!]$ then $\Gamma \vDash t_1 \sqsubseteq^{obs} t_2 : A$*

4.2 Logical Relation

Observational equivalence and approximation are extremely difficult to prove directly, so we use the usual method of proving observational results by using a *logical relation* that we prove *sound* with respect to observational approximation. Due to the non-well-founded nature of recursive types (and the dynamic type specifically), we develop a *step-indexed* logical relation following Ahmed [2006]. We define our logical relation for error approximation in Figure 12. Because our notion of error approximation is not the standard notion of approximation, the definition is a bit unusual, but this is necessary for technical reasons.

It is key to compositional reasoning about embedding-projection pairs that approximation be *transitive* and care must be taken to show transitivity for a step-indexed relation. However, for standard definitions of logical relations for observational equivalence, it is difficult to prove transitivity directly. Therefore, it is often established through indirect reasoning—e.g., by setting up a biorthogonal ($\top\top$-closed) logical relation so one can easily show it is complete with respect to observational equivalence, which in turn implies that it must be transitive since observational equivalence is easily proven transitive. The reason establishing transitivity is tricky is that a step-indexed relation is *not* transitive at a *fixed* index, i.e., if $e_1 \sim^i e_2$ and $e_2 \sim^i e_3$ it is not necessarily the case that $e_1 \sim^i e_3$. For instance, $e_1 \sim^i e_2$ might be related because e_1 terminates in less than i steps and has the same behavior as e_2 which takes more than i steps to terminate, whereas $e_2 \sim^i e_3$ are related because they both take i steps of reduction so cannot be distinguished in i steps but have different behavior when run for more steps. One direct method for proving transitivity, originally presented in Ahmed [2006], is to observe that two terms are observationally equivalent when each divergence approximates the other, and then use a step-indexed relation for divergence approximation. Because a conjunction of transitive relations is transitive, this proves transitivity of equivalence. A step-indexed relation for divergence approximation can be shown to have a kind of "half-indexed" transitivity, i.e., if $e_1 \prec^i e_2$ and for every natural j, we know $e_2 \prec^j e_3$ then $e_1 \prec^i e_3$. We have a similar issue with error approximation: the naïve logical relation for error approximation is not clearly transitive. Inspired by the case of observational equivalence, we similarly "split" our logical relation into two relations that can be proven transitive by an argument similar to divergence approximation. However, unlike for observational equivalence, our two relations are not the same. Instead, one $\sqsubseteq\prec$ is error approximation up to divergence on the *left* and the other $\sqsubseteq\succ$ is error approximation up to divergence on the *right*.

For a given natural number $i \in \mathbb{N}$ and type A, and *closed terms* t_1, t_2 of type A, $t_1 \sqsubseteq\prec^i_{t,A} t_2$ intuitively means that, if we only inspect t_1's behavior up to i uses of **unroll** $\cdot$, then it appears that t_1 error approximates t_2. Less constructively, it means that we cannot show that t_1 does *not* error approximate t_2 when limited to i uses of **unroll** $\cdot$. However, even if we knew $t_1 \sqsubseteq\prec^i_{t,A} t_2$ for *every* $i \in \mathbb{N}$, it still might be the case that t_1 diverges, since no finite number of unrolling can ever exhaust t_1's behavior. So we also require that we know $t_1 \sqsubseteq\succ^i_{t,A} t_2$, which means that up to i uses of unroll on t_2, it appears that t_1 error approximates t_2.

The above intuition should help to understand the definition of error approximation for terms (i.e., the relations $\sqsubseteq\prec_t$ and $\sqsubseteq\succ_t$). The relation $t_1 \sqsubseteq\prec^i_{t,A} t_2$ is defined by inspection of t_1's behavior: it holds if t_1 is still running after $i+1$ unrolls; or if it steps to an error in fewer than i unrolls; or if it results in a value in fewer than i unrolls and also t_2 runs to a value and those values are related for the remaining steps. The definition of $t_1 \sqsubseteq\succ^i_{t,A} t_2$ is defined by inspection of t_2's behavior: it holds if t_2 is still running after $i + 1$ unrolls; or if t_2 steps to an error in fewer than i steps then t_1 errors as well; or if t_2 steps to a value, either t_1 errors or steps to a value related for the remaining steps.

While the relations and $\sqsubseteq\succ_t$ on terms are different, fortunately, the relations on values are essentially the same, so we abstract over the cases by having the symbol $\sqsubseteq\sim$ to range over either $\sqsubseteq\prec$ or $\sqsubseteq\succ$. For values of recursive type, if the step-index is 0, we consider them related, because otherwise we would need to perform an unroll to inspect them further. Otherwise, we decrement the index and check if they are related. Decrementing the index here is exactly what makes the definition of the relation well-founded. For the standard types, the value relation definition is indeed standard: pairs are related when the two sides are related, sums must be the same case and functions must be related when applied to any related values in the future (i.e., when we may have exhausted some of the available steps).

Finally, we extend these relations to *open* terms in the standard way: we define substitutions to be related pointwise (similar to products) and then say that $\Gamma \vDash t_1 \sqsubseteq\sim t_2 : A$ holds if for every pair of substitutions γ_1, γ_2 related for i steps, the terms after substitution, written $t_1[\gamma_1]$ and $t_2[\gamma_2]$,

$$
\begin{array}{rcl}
\sqsubseteq<^{i}_{t,A}, \sqsubseteq>^{i}_{t,A} & \subseteq & \{t \mid \cdot \vdash t : A\}^2 \\
t_1 \sqsubseteq<^{i}_{t,A} t_2 & \overset{\text{def}}{=} & (\exists t_1'.\, t_1 \Mapsto^{i+1} t_1') \\
 & & \vee (\exists j \leq i.\, t_1 \Mapsto^{j} \mho) \\
 & & \vee (\exists j \leq i, v_1 \sqsubseteq<^{i-j}_{v,A} v_2.\, t_1 \Mapsto^{j} v_1 \wedge t_2 \Mapsto^{*} v_2) \\
t_1 \sqsubseteq>^{i}_{t,A} t_2 & \overset{\text{def}}{=} & (\exists t_2'.\, t_2 \Mapsto^{i+1} t_2') \\
 & & \vee (\exists j \leq i.\, t_2 \Mapsto^{j} \mho \wedge t_1 \Mapsto^{*} \mho) \\
 & & \vee (\exists j \leq i, v_2.\, t_2 \Mapsto^{j} v_2 \wedge \\
 & & \qquad (t_1 \Mapsto^{*} \mho \vee \exists v_1.\, t_1 \Mapsto^{*} v_1 \wedge v_1 \sqsubseteq>^{i-j}_{v,A} v_2)) \\
 & & \\
\sqsubseteq<^{i}_{v,A}, \sqsubseteq>^{i}_{v,A} & \subseteq & \{v \mid \cdot \vdash v : A\}^2 \qquad \text{where } \sqsubseteq\sim \,\in \{\sqsubseteq<^{\cdot}_{\cdot,\cdot}, \sqsubseteq>^{\cdot}_{\cdot,\cdot}\} \\
v_1 \sqsubseteq\sim^{0}_{v,\mu\alpha.A} v_2 & \overset{\text{def}}{=} & \top \\
\mathbf{roll}_{\mu\alpha.A}\, v_1 \sqsubseteq\sim^{i+1}_{v,\mu\alpha.A} \mathbf{roll}_{\mu\alpha.A}\, v_2 & \overset{\text{def}}{=} & v_1 \sqsubseteq\sim^{i}_{v,A[\alpha\mapsto\mu\alpha.A]} v_2 \\
\langle\rangle \sqsubseteq\sim^{i}_{v,1} \langle\rangle & \overset{\text{def}}{=} & \top \\
\langle v_1, v_1'\rangle \sqsubseteq\sim^{i}_{v,A\times A'} \langle v_2, v_2'\rangle & \overset{\text{def}}{=} & v_1 \sqsubseteq\sim^{i}_{v,A} v_2 \wedge v_1' \sqsubseteq\sim^{i}_{v,A'} v_2' \\
v_1 \sqsubseteq\sim^{i}_{v,A+B} v_2 & \overset{\text{def}}{=} & (\exists (v_1' \sqsubseteq\sim^{i}_{v,A} v_2') \wedge v_1 = \mathbf{inj}\, v_1' \wedge v_2 = \mathbf{inj}\, v_2') \\
 & & \vee (\exists (v_1' \sqsubseteq\sim^{i}_{v,B} v_2') \wedge v_1 = \mathbf{inj}'\, v_1' \wedge v_2 = \mathbf{inj}'\, v_2') \\
v_1 \sqsubseteq\sim^{i}_{v,A\to B} v_2 & \overset{\text{def}}{=} & \forall j \leq i. \forall (v_1' \sqsubseteq\sim^{j}_{v,A} v_2').\, v_1\, v_1' \sqsubseteq\sim^{i}_{t,B} v_2\, v_2' \\
 & & \\
\cdot \sqsubseteq\sim^{i}_{v,\cdot}\, \cdot & \overset{\text{def}}{=} & \top \\
\gamma_1, v_1/x \sqsubseteq\sim^{i}_{v,\Gamma,x:A} \gamma_2, v_2/x & \overset{\text{def}}{=} & \gamma_1 \sqsubseteq\sim^{i}_{v,\Gamma} \gamma_2 \wedge v_1 \sqsubseteq\sim^{i}_{v,A} v_2 \\
 & & \\
\Gamma \vDash t_1 \sqsubseteq\sim t_2 : A & \overset{\text{def}}{=} & \forall i \in \mathbb{N}, (\gamma_1 \sqsubseteq\sim^{i}_{v,\Gamma} \gamma_2).\, t_1[\gamma_1] \sqsubseteq\sim^{i}_{t,A} t_2[\gamma_2] \\
 & & \\
\Gamma \vDash t_1 \sqsubseteq t_2 : A & \overset{\text{def}}{=} & \Gamma \vdash t_1 \sqsubseteq< t_2 : A \wedge \Gamma \vdash t_1 \sqsubseteq> t_2 : A
\end{array}
$$

Fig. 12. $\lambda_{T,\mho}$ Error Approximation Logical Relation

are related for i steps. Then our resulting relation $\Gamma \vDash t_1 \sqsubseteq t_2$ is defined to hold when t_1 error approximates t_2 up to divergence of t_1 ($\sqsubseteq<$), *and* up to divergence of t_2 ($\sqsubseteq>$).

We need the following standard lemmas.

Lemma 4.4 (Downward Closure). *If $j \leq i$ then*

(1) *If $t_1 \sqsubseteq\sim^{i}_{t,A} t_2$ then $t_1 \sqsubseteq\sim^{j}_{t,A} t_2$*

(2) *If $v_1 \sqsubseteq\sim^{i}_{v,A} v_2$ then $v_1 \sqsubseteq\sim^{j}_{v,A} v_2$.*

Lemma 4.5 (Anti-Reduction). *This theorem is different for the two relations as we allow arbitrary steps on the "divergence greater-than" side.*

(1) *If $t_1 \sqsubseteq<^{i}_{t,A} t_2$ and $t_1' \Mapsto^{j} t_1$ and $t_2' \Mapsto^{*} t_2$ then $t_1' \sqsubseteq<^{i+j}_{t,A} t_2'$.*

(2) *If $t_1 \sqsubseteq>^{i}_{t,A} t_2$ and $t_2' \Mapsto^{j} t_2$ and $t_1' \Mapsto^{*} t_1$, then $t_1' \sqsubseteq>^{i+j}_{t,A} t_2'$.*

Lemma 4.6 (Monadic Bind). *For any $i \in \mathbb{N}$, if for any $j \leq i$ and $v_1 \sqsubseteq\sim^{j}_{t,A} v_2$, we can show $E_1[v_1] \sqsubseteq\sim^{j}_{t,A} E_2[v_2]$ holds, then for any $t_1 \sqsubseteq\sim^{i}_{t,A} t_2$, it is the case that $E_1[v_1] \sqsubseteq\sim^{i}_{t,A} E_2[v_2]$.*

We then prove that our logical relation is sound for observational error approximation by showing that it is a congruence relation (see the extended version [New and Ahmed 2018])and showing that if we can prove error approximation up to divergence on the left *and* on the right, then we have true error approximation.

Theorem 4.7 (Logical Relation implies Observational Error Approximation). *If* $\Gamma \vDash t_1 \sqsubseteq t_2 : A$*, then* $\Gamma \vDash t_1 \sqsubseteq^{obs} t_2 : A$

4.3 Approximation and Equivalence Lemmas

The step-indexed logical relation is on the face of it quite complex, especially due to the splitting of error approximation into two step-indexed relations. However, we should view the step-indexed relation as an "implementation" of the high-level concept of error approximation, and we work as much as possible with the error approximation relation $\Gamma \vDash t_1 \sqsubseteq t_2 : A$. In order to do this we now prove some high-level lemmas, which are proven using the step-indexed relations, but allow us to develop conceptual proofs of the key theorems of the paper.

First, there is reflexivity, also known as the *fundamental lemma*, which is proved using the same congruence cases as the soundness theorem (theorem 4.7.) Note that by the definition of our logical relation, this is really a kind of *monotonicity* theorem for every term in the language, the first component of our graduality proof.

Corollary 4.8 (Reflexivity). *If* $\Gamma \vdash t : A$ *then* $\Gamma \vDash t \sqsubseteq t : A$

Next, crucial to reasoning about ep pairs is the use of *transitivity*, a notoriously tedious property to prove for step-indexed logical relations. This is where our splitting of error-approximation into two pieces proves essential, adapting the approach for divergence-approximation relations introduced in Ahmed [2006]. The proof works as follows: due to the function and open-term cases, we cannot simply prove transitivity in the limit directly. Instead we get a kind of "asymmetric" transitivity: if $t_1 \sqsubseteq\prec^{i}_{t,A} t_2$ and *for any* $j \in \mathbb{N}$, $t_2 \sqsubseteq\prec^{j}_{t,A} t_3$, then we know $t_1 \sqsubseteq\prec^{i}_{t,A} t_3$. We abbreviate the $\forall j$ part as $t_2 \sqsubseteq\prec^{\omega}_{t,A} t_3$ in what follows. The key to the proof is in the function and open terms cases, which rely on reflexivity, corollary 4.8, as in Ahmed [2006]. Reflexivity says that when we have $v_1 \sqsubseteq\prec^{i}_{v,A} v_2$ then we also have $v_2 \sqsubseteq\prec^{\omega}_{v,A} v_2$, which allows us to use the inductive hypothesis.

Lemma 4.9 (Transitivity for Closed Terms/Values). *The following are true for any* A.

(1) *If* $t_1 \sqsubseteq\prec^{i}_{t,A} t_2$ *and* $t_2 \sqsubseteq\prec^{\omega}_{t,A} t_3$ *then* $t_1 \sqsubseteq\prec^{i}_{t,A} t_3$.
(2) *If* $v_1 \sqsubseteq\prec^{i}_{t,A} v_2$ *and* $v_2 \sqsubseteq\prec^{\omega}_{t,A} v_3$ *then* $v_1 \sqsubseteq\prec^{i}_{t,A} v_3$.

Similarly,

(1) *If* $t_1 \sqsubseteq\succ^{\omega}_{t,A} t_2$ *and* $t_2 \sqsubseteq\succ^{i}_{t,A} t_3$ *then* $t_1 \sqsubseteq\succ^{i}_{t,A} t_3$.
(2) *If* $v_1 \sqsubseteq\succ^{\omega}_{t,A} v_2$ *and* $v_2 \sqsubseteq\succ^{i}_{t,A} v_3$ *then* $v_1 \sqsubseteq\succ^{i}_{t,A} v_3$.

Lemma 4.10 (Transitivity). *If* $\Gamma \vdash t_1 \sqsubseteq t_2 : A$ *and* $\Gamma \vdash t_2 \sqsubseteq t_3 : A$ *then* $\Gamma \vdash t_1 \sqsubseteq t_3 : A$.

Next, we want to extract approximation and equivalence principles for *open* programs from syntactic operational properties of *closed* programs. First, obviously any operational reduction is a contextual equivalence, and the next lemma extends that to open programs. Note that we use $\sqsupseteq\sqsubseteq$ to mean approximation in both directions, i.e., equivalence:

Lemma 4.11 (Open β Reductions). *Given* $\Gamma \vdash t : A$, $\Gamma \vdash t' : A$, *if for every* $\gamma : \Gamma$, $t[\gamma] \mapsto t'[\gamma]$, *then* $\Gamma \vDash t \sqsupseteq\sqsubseteq t' : A$.

We call this open β reduction because we will use it to justify equivalences that look like an operational reduction, but have open values (i.e. including variables) rather than closed as in the operational semantics. For instance,

$$\mathbf{let}\ x = y\ \mathbf{in}\ t \sqsupseteq\sqsubseteq t[y/x]$$

$$\begin{array}{rcl} E[\mathrm{let}\, x = t\, \mathrm{in}\, s] & \sqsupseteq\sqsubseteq & \mathrm{let}\, x = t\, \mathrm{in}\, E[s] \\ E[\mathrm{let}\, \langle x, y\rangle = t\, \mathrm{in}\, s] & \sqsupseteq\sqsubseteq & \mathrm{let}\, \langle x, y\rangle = t\, \mathrm{in}\, E[s] \\ E[\mathrm{case}\, t\, \mathrm{of}\ \ \mathrm{inj}\, x.\, s \mid \mathrm{inj}'\, x'.\, s'] & \sqsupseteq\sqsubseteq & \mathrm{case}\, t\, \mathrm{of}\ \ \mathrm{inj}\, x.\, E[s] \mid \mathrm{inj}'\, x'.\, E[s'] \end{array}$$

Fig. 13. Commuting Conversions

and

$$\mathrm{let}\, \langle x, y\rangle = \langle x', y'\rangle\, \mathrm{in}\, t \sqsupseteq\sqsubseteq t[x'/x, y'/y]$$

Additionally, it is convenient to use η expansions for our types as well. Note that since we are using a call-by-value language, the η expansion for functions is restricted to *values*.

Lemma 4.12 (η Expansion).

(1) *For any* $\Gamma \vdash v : A \to B$, $v \sqsupseteq\sqsubseteq \lambda(x : A).\, v\, x$
(2) *For any* $\Gamma, x : A + A', \Gamma' \vdash t : B$,

$$t \sqsupseteq\sqsubseteq \mathrm{case}\, x\, \mathrm{of}\ \ \mathrm{inj}\, y.\, t[\mathrm{inj}\, y'/x] \mid \mathrm{inj}'\, y'.\, t[\mathrm{inj}'\, y'/x]$$

(3) *For any* $\Gamma, x : A \times A', \Gamma' \vdash t : B$,

$$t \sqsupseteq\sqsubseteq \mathrm{let}\, \langle y, y'\rangle = x\, \mathrm{in}\, t[\langle y, y'\rangle/x]$$

Next, with term constructors that involve continuations, we often need to rearrange the programs such as the “case-of-case” transformation. These are called commuting conversions and are presented in Figure 13.

Lemma 4.13 (Commuting Conversions). *All of the commuting conversions in Figure 13 are equivalences.*

Next, the following theorem is the main reason we so heavily use *evaluation contexts*. It is a kind of open version of the monadic bind lemma lemma 4.6.

Lemma 4.14 (Evaluation contexts are linear). *If* $\Gamma \vdash t : A$ *and* $\Gamma, x : A \vdash E[x] : B$, *then*

$$\mathrm{let}\, x = t\, \mathrm{in}\, E[x] \sqsupseteq\sqsubseteq E[t]$$

The concepts of pure and terminating terms are useful because when subterms are pure or terminating, they can be moved around to prove equivalences more easily.

Definition 4.15 (Pure, Terminating Terms). (1) A term $\Gamma \vdash t : A$ is *terminating* if for any closing γ, either $t[\gamma] \Mapsto \mho$ or $t[\gamma] \Mapsto v$ for some v.
(2) A term $\Gamma \vdash t : A$ is *pure* if for any closing γ, $t[\gamma] \Mapsto v$ for some v.

Lemma 4.16 (Pure Terms are Essentially Values). *If* $\Gamma \vdash t : A$ *is a pure term, then for any* $\Gamma, x : A \vdash s : B$, $\mathrm{let}\, x = t\, \mathrm{in}\, s \sqsupseteq\sqsubseteq s[t/x]$ *holds.*

Also, since we consider all type errors to be equal, terminating terms can be reordered:

Lemma 4.17 (Terminating Terms Commute). *If* $\Gamma \vdash t : A$ *and* $\Gamma \vdash t' : A'$ *and* $\Gamma, x : A, y : A' \vdash s : B$, *then* $\mathrm{let}\, x = t\, \mathrm{in}\, \mathrm{let}\, x' = t'\, \mathrm{in}\, s \sqsupseteq\sqsubseteq \mathrm{let}\, x' = t'\, \mathrm{in}\, \mathrm{let}\, x = t\, \mathrm{in}\, s$

5 CASTS AS EMBEDDING-PROJECTION PAIRS

In this section, we show how arbitrary casts can be broken down into *embedding-projection* pairs. First, we define type dynamism and show that casts between less and more dynamic types form an ep pair. Then we will show that every cast is a composition of an upcast and a downcast.

5.1 Embedding-Projection Pairs

First, we define ep pairs with respect to *logical* approximation. Note that since logical approximation implies observational error approximation, these are also ep pairs with respect to observational error approximation. However, in theorems where we *construct* new ep pairs from old ones, we will need that the input ep pairs are logical ep pairs, not just observational, because we have not proven that logical approximation is *complete* for observational error approximation. As with casts, we use evaluation contexts for convenience.

Definition 5.1 (EP Pair). A (logical) ep pair $(E_e, E_p) : A \triangleleft B$ is a pair of an *embedding* $E_e[\cdot : A] : B$ and a *projection* $E_p[\cdot : B] : A$ satisfying

(1) Retraction: $x : A \vdash x \sqsupseteq\sqsubseteq E_p[E_e[x]] : A$
(2) Projection: $y : B \vdash E_e[E_p[y]] \sqsubseteq y : B$

Next, we prove that in any embedding-projection pair that embeddings are pure (always produce a value with no effects) and that projections are terminating (either error or produce a value). Paired with the lemmas we have proven about pure and terminating programs in the previous section, we will be able to prove theorems about ep pairs more easily.

Lemma 5.2 (Embeddings are Pure). *If $E_e, E_p : A \triangleleft B$ is an embedding-projection pair then $x : A \vdash E_e[x] : B$ is pure.*

Proof. The ep pair property states that $x : A \vDash E_p[E_e[x]] \sqsupseteq\sqsubseteq x : A$ Given any value $\cdot \vdash v : A$, by Lemma 4.8, we know $v \sqsubseteq\prec^{0}_{t,A} E_p[E_e[v]]$ and since $v \mapsto^{0} v$, this means there exists v' such that $E_p[E_e[v]] \mapsto v'$, and since E_p is an evaluation context, this means there must exist v'' with $E_p[E_e[v]] \mapsto E_p[v''] \mapsto v'$. □

Lemma 5.3 (Projections are Terminating). *If $E_e, E_p : A \triangleleft B$ is an embedding-projection pair then $y : B \vdash E_p[y] : A$ is terminating.*

Proof. The ep pair property states that $y : B \vDash E_e[E_p[y]] \sqsubseteq y : B$ Given any $v : B$, by Lemma 4.8, we know $E_e[E_p[v]] \sqsubseteq\succ^{0}_{t,B} v$ so therefore either $E_e[E_p[v]] \mapsto \mho$, which because E_e is pure means $E_p[v] \mapsto \mho$, or $E_e[E_p[v]] \mapsto v'$ which by strictness of evaluation contexts means $E_p[v] \mapsto v''$ for some v''. □

Crucially, ep pairs can be constructed using simple function composition. First, the identity function is an ep pair by reflexivity.

Lemma 5.4 (Identity EP Pair). *For any type A, $[\cdot], [\cdot] : A \triangleleft A$.*

Second, if we compose the embeddings one way and projections the opposite way, the result is an ep pair, by congruence.

Lemma 5.5 (Composition of EP Pairs). *For any ep pairs $E_{e,1}, E_{p,1} : A_1 \triangleleft A_2$ and $E_{e,2}, E_{p,2} : A_2 \triangleleft A_3$, $E_{e,2}[E_{e,1}], E_{p,1}[E_{p,2}] : A_1 \triangleleft A_3$.*

5.2 Type Dynamism

Next, we consider type dynamism and its relationship to the casts. The type dynamism relation is presented in Figure 14. The relation $A \sqsubseteq B$ reads as "A is less dynamic than B" or "A is more precise than B". For the purposes of its definition, we can say that it is the least reflexive and transitive relation such that every type constructor is monotone and $?$ is the greatest element. Even the function type is monotone in both input and output argument and for this reason, type dynamism is sometimes called *naïve subtyping* [Wadler and Findler 2009]. However, this gives us

$\boxed{A \sqsubseteq B}$

$$\frac{}{A \sqsubseteq A}\ \textsc{Reflexivity} \qquad \frac{A_1 \sqsubseteq A_2 \qquad A_2 \sqsubseteq A_3}{A_1 \sqsubseteq A_3}\ \textsc{Transitivity} \qquad \frac{}{A \sqsubseteq ?}\ \textsc{Dyn-Top}$$

$$\frac{A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{A_1 + B_1 \sqsubseteq A_2 + B_2}\ \textsc{Sum} \qquad \frac{A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{A_1 \times B_1 \sqsubseteq A_2 \times B_2}\ \textsc{Prod} \qquad \frac{A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{A_1 \to B_1 \sqsubseteq A_2 \to B_2}\ \textsc{Fun}$$

Fig. 14. Type Dynamism

$\boxed{c : A \sqsubseteq B}$

$$\frac{A \in \{1, ?\}}{id(A) : A \sqsubseteq A} \qquad \frac{A \neq ? \qquad c : A \sqsubseteq \lfloor A \rfloor \qquad \dfrac{}{tag(\lfloor A \rfloor) : \lfloor A \rfloor \sqsubseteq ?}}{tag(\lfloor A \rfloor) \circ c : A \sqsubseteq ?}$$

$$\frac{c : A_1 \sqsubseteq A_2 \qquad d : B_1 \sqsubseteq B_2}{c \times d : A_1 \times B_1 \sqsubseteq A_2 \times B_2} \qquad \frac{c : A_1 \sqsubseteq A_2 \qquad d : B_1 \sqsubseteq B_2}{c + d : A_1 + B_1 \sqsubseteq A_2 + B_2} \qquad \frac{c : A_1 \sqsubseteq A_2 \qquad d : B_1 \sqsubseteq B_2}{c \to d : A_1 \to B_1 \sqsubseteq A_2 \to B_2}$$

Fig. 15. Canonical Proof Terms for Type Dynamism

no *semantic* intuition about what it could possibly mean. We propose that $A \sqsubseteq B$ should hold when the casts between A and B form an embedding-projection pair $E_{\langle B \Leftarrow A \rangle}, E_{\langle A \Leftarrow B \rangle} : A \triangleleft B$. We can then view each of the cases of the gradual guarantee as being *compositional rules* for constructing ep pairs. Reflexivity and transitivity correspond to the identity and composition of ep pairs, and the monotonicity of types comes from the fact that *every* functor preserves ep pairs.

Taking this idea further, we can view type dynamism not just as an *analysis* of pre-existing gradual type casts, but by considering its simple *proof theory*, we can view proofs of type dynamism as *synthesizing* the definitions of casts. To accomplish this, we give a *refined* formulation of the proof theory of type dynamism in Figure 15, which includes explicit proof terms $c : A \sqsubseteq B$. The methodology behind the presentation is to make reflexivity, transitivity, and the fact that ? is a greatest element into *admissible* properties of the system, rather than primitive rules. First, by making proofs admissible, we see in detail how bigger casts are built up from small pieces.

Second, this formulation satisfies a *canonicity* property: there is exactly one proof of any given derivation, which simplifies the definition of the semantics. By giving a presentation where derivations are canonical, the typical "coherence" theorem, that says any two derivations have equivalent semantics, becomes trivial. An alternative formulation would define an ep-pair semantics where reflexivity and transitivity denote identity and composition of ep pairs, and then prove that any two derivations have equivalent semantics. Instead, we define admissible constructions for reflexivity and transitivity, and then prove a *decomposition lemma* (lemma 5.11) that states that the ep-pair semantics interprets our admissible reflexivity derivation as identity and transitivity derivation as a composition. In short, our presentation makes it obvious that the semantics is coherent, but not that it is built out of composition, whereas the alternative makes it obvious that the semantics is built out of composition, but not that it is coherent.

We present the proof terms for type dynamism in Figure 15. As in presentations of sequent calculus, we include the identity ep pair (reflexivity) only for the base types $1, ?$. The next rule $tag(\lfloor A \rfloor) \circ c : A \sqsubseteq ?$ states that any casts between a non-dynamic type A and the dynamic type $?$ are the composition $\circ$ of, first, a tagging-untagging ep-pair with its underlying tag type $tag(\lfloor A \rfloor) : \lfloor A \rfloor \sqsubseteq ?$ and an ep pair from A to its tag type $c : A \sqsubseteq \lfloor A \rfloor$. The product, sum, and function rules are written to evoke that their ep pairs use the functorial action.

As mentioned the proof terms are *canonical*, meaning there is at most one derivation of any $A \sqsubseteq B$.

$$
\begin{array}{rcl@{\qquad}rcl}
id(?) & \overset{\text{def}}{=} & id(?) & (tag(\lfloor A \rfloor) \circ c) \circ d & \overset{\text{def}}{=} & tag(\lfloor A \rfloor) \circ (c \circ d) \\
id(1) & \overset{\text{def}}{=} & id(1) & (id(A)) \circ d & \overset{\text{def}}{=} & d \\
id(A_1 \times A_2) & \overset{\text{def}}{=} & id(A_1) \times id(A_2) & (c \times d) \circ (c' \times d') & \overset{\text{def}}{=} & (c \circ c') \times (d \circ d') \\
id(A_1 + A_2) & \overset{\text{def}}{=} & id(A_1) + id(A_2) & (c + d) \circ (c' + d') & \overset{\text{def}}{=} & (c \circ c') + (d \circ d') \\
id(A_1 \to A_2) & \overset{\text{def}}{=} & id(A_1) \to id(A_2) & (c \to d) \circ (c' \to d') & \overset{\text{def}}{=} & (c \circ c') \to (d \circ d')
\end{array}
$$

$$
\begin{array}{rcl}
top(?) & \overset{\text{def}}{=} & id(?) \\
top(1) & \overset{\text{def}}{=} & tag(1) \circ id(1) \\
top(A \times B) & \overset{\text{def}}{=} & tag(? \times ?) \circ (top(A) \times top(B)) \\
top(A + B) & \overset{\text{def}}{=} & tag(? + ?) \circ (top(A) + top(B)) \\
top(A \to B) & \overset{\text{def}}{=} & tag(? \to ?) \circ (top(A) \to top(B))
\end{array}
$$

Fig. 16. Type Dynamism Admissible Proof Terms

$$
\begin{array}{rcl@{\qquad}rcl}
m \in \{e, p\} & & & & & \\
\overline{e} & \overset{\text{def}}{=} & p & E_{e,tag(G) \circ d} & \overset{\text{def}}{=} & E_{e,tag(G)}[E_{e,d}] \\
\overline{p} & \overset{\text{def}}{=} & e & E_{p,tag(G) \circ d} & \overset{\text{def}}{=} & E_{p,d}[E_{p,tag(G)}] \\
E_{m,id(A)} & \overset{\text{def}}{=} & [\cdot] & E_{m,c \times c'} & \overset{\text{def}}{=} & E_{m,c} \times E_{m,c'} \\
E_{e,tag(G)} & \overset{\text{def}}{=} & \mathbf{roll}_{[\![?]\!]}\ \mathbf{inj}_G\ [\cdot] & E_{m,c+c'} & \overset{\text{def}}{=} & E_{m,c} + E_{m,c'} \\
E_{p,tag(G)} & \overset{\text{def}}{=} & \mathbf{case\ unroll}\ [\cdot]\ \mathbf{of\ inj}_G\ x. x \mid \mathbf{else}. \mho & E_{m,c \to c'} & \overset{\text{def}}{=} & E_{\overline{m},c} \to E_{m,c'}
\end{array}
$$

Fig. 17. Type Dynamism Cast Translation

Lemma 5.6 (Canonical Type Dynamism Derivations). *Any two derivations* $c, d : A \sqsubseteq B$ *are equal* $c = d$.

Next, we need to show that the rules in Figure 14 are all *admissible* in the refined system Figure 15. The proof of admissibility is given in Figure 16. First, to show reflexivity is admissible, we construct the proof $id(A) : A \sqsubseteq A$. It is primitive for ? and 1 and we use the congruence rule to lift the others. Second, to show transitivity is admissible, for every $d : A_1 \sqsubseteq A_2$ and $c : A_2 \sqsubseteq A_3$, we construct their *composite* $c \circ d : A_1 \sqsubseteq A_3$ by recursion on c. If c is a primitive composite with a tag, we use associativity of composition to push the composite in. If c is the identity, the composite is just d. Otherwise, both c and d must be between a connective, and we push the compositions in. Finally, we show that ? is the most dynamic type by constructing a derivation $top(A) : A \sqsubseteq ?$ for every A. For ?, it is just the identity; for the remaining types, we use the tag ep pair and compose with lifted uses of *top*.

Next, we construct a *semantics* for the type dynamism proofs that justifies the intuition we have given so far; it is presented in Figure 17. Every type dynamism proof $c : A \sqsubseteq B$ defines a *pair* of an embedding $E_{e,c}$ and a projection $E_{p,c}$. Since many rules are the same for embeddings and projections, we use $m \in \{e, p\}$ to abstract over the *mode* of the cases. We define the *complement* of a mode $\overline{m}$ to swap between embeddings and projections; it is used in the function case. The primitive identity casts are interpreted as the identity, and the primitive composition of casts $tag(G) \circ d$ is interpreted as the composition of ep pairs of $tag(G)$ and d. The tag type derivation is interpreted by the same definition as the cast in Figure 10: tagging puts the correct sum case and $\mathbf{roll}_?$, and untagging unwraps if its the correct sum case and otherwise errors. We abbreviate this as pattern matching with an "else" clause, where the else clause stands for all of the clauses that do not match the tag type G. The desugaring to repeated case statements on sums should be clear. The product and sum type are just given by their functorial action with the same mode. The function type similarly uses its functorial action, but swaps from embedding to projection or vice-versa on the domain side. This shows that there is nothing strange about the function rule: it is the same construction as for subtyping, but constructing arrows back and forth at the same time. The fact that contravariant functors are covariant with respect to ep pairs in this way is *precisely* the reason they are used extensively in domain theory.

We next verify that these actually are embedding-projection pairs. To do this, we use the identity and composition lemmas proved before, but we also need to use *functoriality* of the actions of type constructors, meaning that the action of the type interacts well with identity and composition of evaluation contexts.

Lemma 5.7 (Identity Extension).

$$[\cdot] \times [\cdot] \sqsupseteq\sqsubseteq [\cdot] \quad \textit{and} \quad [\cdot] + [\cdot] \sqsupseteq\sqsubseteq [\cdot] \quad \textit{and} \quad [\cdot] \to [\cdot] \sqsupseteq\sqsubseteq [\cdot]$$

In a call-by-value language, the functoriality rules do not hold in general for the product functor, but they do for terminating programs because their order of evaluation is irrelevant. Also notice that when composing using the functorial action of the function type $\to$, the composition flips on the domain side, because the function type is *contravariant* in its domain.

Lemma 5.8 (Functoriality for Terminating Programs). *The following equivalences are true for any well-typed,* terminating *evaluation contexts.*

$$\begin{array}{rcl}(E_2 \times E_2')[E_1 \times E_1'] & \sqsupseteq\sqsubseteq & (E_2[E_1]) \times (E_2'[E_1']) \\ (E_2 + E_2')[E_1 + E_1'] & \sqsupseteq\sqsubseteq & (E_2[E_1]) + (E_2'[E_1']) \\ (E_2 \to E_2')[E_1 \to E_1'] & \sqsupseteq\sqsubseteq & (E_1[E_2]) \to (E_2'[E_1'])\end{array}$$

With these cases covered, we can show the casts given by type dynamism really are ep pairs.

Lemma 5.9 (Type Dynamism Derivation denotes EP Pair). *For any derivation $c : A \sqsubseteq B$, then $E_{e,c}, E_{p,c} : [\![A]\!] \triangleleft [\![B]\!]$ are an ep pair.*

Next, while we showed that transitivity and reflexivity were admissible with the $id(A)$ and $c \circ d$ definitions, their semantics are not given directly by the identity and composition of evaluation contexts. We justify this notation by the following theorems. First, $id(A)$ is the identity by identity extension.

Lemma 5.10 (Reflexivity Proofs denote Identity). *For every A, $E_{e,id(A)} \sqsupseteq\sqsubseteq [\cdot]$ and $E_{p,id(A)} \sqsupseteq\sqsubseteq [\cdot]$.*

Second, we have our key *decomposition* theorem. While the composition theorem says that the composition of any two ep pairs is an ep pair, the *decomposition* theorem is really a theorem about the *coherence* of our type dynamism proofs. It says that given any ep pair given by $c : A_1 \sqsubseteq A_3$, if we can find a middle type A_2, then we can *decompose* c's ep pairs into a composition. This theorem is used extensively, especially in the proof of the gradual guarantee.

Lemma 5.11 (Decomposition of Upcasts, Downcasts). *For any derivations $c : A_1 \sqsubseteq A_2$ and $c' : A_2 \sqsubseteq A_3$, the upcasts and downcasts given by their composition $c' \circ c$ are equivalent to the composition of their casts given by c, c':*

$$x : [\![A_1]\!] \vDash E_{e,c' \circ c}[x] \sqsupseteq\sqsubseteq E_{e,c'}[E_{e,c}[x]] : [\![A_3]\!]$$

$$y : [\![A_3]\!] \vDash E_{p,c' \circ c}[y] \sqsupseteq\sqsubseteq E_{p,c}[E_{p,c'}[y]] : [\![A_1]\!]$$

Finally, now that we have established the meaning of type dynamism derivations and proven the decomposition theorem, we can dispense with direct manipulation of derivations. So we define the following notation for ep pairs that just uses the types:

Definition 5.12 (EP Pair Semantics). Given $c : A \sqsubseteq B$, we define $E_{m,A,B} = E_{m,c}$.

5.3 Casts Factorize Into EP Pairs

Next, we show how the upcasts and downcasts are sufficient to construct all the casts of λ_G.

First, when $A \sqsubseteq B$, the ep pair semantics and the cast semantics coincide:

Lemma 5.13 (Upcasts and Downcasts are Casts). *If* $A \sqsubseteq B$ *then* $E_{\langle B \Leftarrow A\rangle} \sqsupseteq\sqsubseteq E_{e,A,B}$ *and* $E_{\langle A \Leftarrow B\rangle} \sqsupseteq\sqsubseteq E_{p,A,B}$.

Next, we show that the "general" casts of the gradual language can be *factorized* into a composition of an upcast followed by a downcast. First, we show that factorizing through any type is equivalent to factorizing through the dynamic type, as a consequence of the *retraction* property of ep pairs.

Lemma 5.14 (Any Factorization is equivalent to Dynamic). *For any* A_1, A_2, A' *with* $A_1 \sqsubseteq A'$ *and* $A_2 \sqsubseteq A'$, $E_{p,A_2,?}[E_{e,A_1,?}] \sqsupseteq\sqsubseteq E_{p,A_2,A'}[E_{e,A_1,A'}]$.

Proof. By decomposition and the retraction property:

$$E_{p,A_2,?}[E_{e,A_1,?}] \sqsupseteq\sqsubseteq E_{p,A_2,?}[E_{p,A',?}[E_{e,A',?}[E_{e,A_1,?}]]] \sqsupseteq\sqsubseteq E_{p,A_2,A'}[E_{e,A_1,A'}]$$

□

By transitivity of equivalence, this means that factorization through one B is as good as any other. So to prove that every cast factors as an upcast followed by a downcast, we can choose whatever middle type is most convenient. This lets us choose the simplest type possible in the proof. For instance, when factorizing a function cast $\langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle$, we can use the function tag type as the middle type $? \to ?$ and then the equivalence is a simple use of the inductive hypothesis and the functoriality principle.

Lemma 5.15 (Every Cast Factors as Upcast, Downcast). *For any* A_1, A_2, A' *with* $A_1 \sqsubseteq A'$ *and* $A_2 \sqsubseteq A'$, *the cast from* A_1 *to* A_2 *factors through* A': $x : [\![A]\!] \vDash E_{\langle A_2 \Leftarrow A\rangle}[x] \sqsupseteq\sqsubseteq E_{p,A_2,A'}[E_{e,A,A'}[x]] : [\![A_2]\!]$

Proof. (1) If $A_1 \sqsubseteq A_2$, then we choose $A' = A_2$ and we need to show that $E_{\langle A_2 \Leftarrow A_1\rangle} \sqsupseteq\sqsubseteq E_{p,A_2 A_2}[E_{e,A_1 A_2}]$ this follows by lemma 5.13 and lemma 5.10.

(2) If $A_2 \sqsubseteq A_1$, we use a dual argument to the previous case. We choose $A' = A_1$ and we need to show that

$$E_{\langle A_2 \Leftarrow A_1\rangle} \sqsupseteq\sqsubseteq E_{p,A_1 A_2}[E_{e,A_1 A_1}]$$

this follows by lemma 5.13 and lemma 5.10.

(3) $E_{\langle A_2 \times B_2 \Leftarrow A_1 \to B_1\rangle} \overset{\text{def}}{=} E_{\langle A_2 \Leftarrow A_1\rangle} \rightarrow E_{\langle B_2 \Leftarrow B_1\rangle}$ We choose $A' = ? \to ?$. By inductive hypothesis,

$$E_{\langle A_1 \Leftarrow A_2\rangle} \sqsupseteq\sqsubseteq E_{p,A_1,?}[E_{e,A_2,?}] \quad \text{and} \quad E_{\langle B_2 \Leftarrow B_1\rangle} \sqsupseteq\sqsubseteq E_{p,B_2,?}[E_{e,B_1,?}]$$

Then the result holds by functoriality:

$$\begin{aligned} E_{\langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle} &= E_{\langle A_1 \Leftarrow A_2\rangle} \rightarrow E_{\langle B_2 \Leftarrow B_1\rangle} \\ &\sqsupseteq\sqsubseteq (E_{p,A_1,?}[E_{e,A_2,?}]) \rightarrow (E_{p,B_2,?}[E_{e,B_1,?}]) \\ &\sqsupseteq\sqsubseteq (E_{e,A_2,?} \rightarrow E_{p,B_2,?})[E_{p,A_1,?} \rightarrow E_{e,B_1,?}] \\ &= E_{p,A_2 \to B_2,? \to ?}[E_{e,A_1 \to B_2,? \to ?}] \end{aligned}$$

(4) (Products, Sums) Same argument as function case.

(5) $(A_1, A_2 \neq ? \wedge \lfloor A_1 \rfloor \neq \lfloor A_2 \rfloor)$ $E_{\langle A_2 \Leftarrow A_1\rangle} \overset{\text{def}}{=} \mathbf{let}\ x = [\cdot]\ \mathbf{in}\ \mho$ We choose $A' = ?$, so we need to show: $\mathbf{let}\ x = [\cdot]\ \mathbf{in}\ \mho \sqsupseteq\sqsubseteq E_{p,A_2,?}[E_{e,A_1,?}]$. By embedding, projection decomposition this is equivalent to

$$\mathbf{let}\ x = [\cdot]\ \mathbf{in}\ \mho \sqsupseteq\sqsubseteq E_{p,A_2,\lfloor A_2 \rfloor}[E_{p,\lfloor A_2 \rfloor,?}[E_{e,A_1,\lfloor A_1 \rfloor}[E_{e,A_1,\lfloor A_1 \rfloor}]]]$$

Which holds by open β because the embedding $E_{e,A_1,\lfloor A_1 \rfloor}$ is pure and $\lfloor A_1 \rfloor \neq \lfloor A_2 \rfloor$.

□

$\boxed{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2}$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \langle B_1 \Leftarrow A_1 \rangle t_1 \sqsubseteq \langle B_2 \Leftarrow A_2 \rangle t_2 : B_1 \sqsubseteq B_2} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2 \qquad A_1 \sqsubseteq A_2 \qquad \Gamma_1' \sqsubseteq \Gamma_2'}{\Gamma_1, x_1 : A_1, \Gamma_2 \sqsubseteq \Gamma_2, x_2 : A_2, \Gamma_2' \vdash x_1 \sqsubseteq x_2 : A_1 \sqsubseteq A_2}$$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \langle\rangle \sqsubseteq \langle\rangle : 1 \sqsubseteq 1} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad \Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1' \sqsubseteq t_2' : A_1' \sqsubseteq A_2'}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \langle t_1, t_1' \rangle \sqsubseteq \langle t_2, t_2' \rangle : A_1 \times A_1' \sqsubseteq A_2 \times A_2'}$$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \times A_1' \sqsubseteq A_2 \times A_2' \qquad \Gamma_1, x_1 : A_1, x_1' : A' \sqsubseteq \Gamma_2, x_2 : A_2, x_2' : A_2' \vdash t_1' \sqsubseteq t_2' : B_1 \sqsubseteq B_2}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \mathsf{let}\ \langle x_1 : A_1, x_1' : A' \rangle = t_1\ \mathsf{in}\ t_1' \sqsubseteq \mathsf{let}\ \langle x_2 : A_2, x_2' : A_2' \rangle = t_2\ \mathsf{in}\ t_2' : B_1 \sqsubseteq B_2}$$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad A_1' \sqsubseteq A_2'}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \mathsf{inj}\, t_1 \sqsubseteq \mathsf{inj}\, t_2 : A_1 + A_1' \sqsubseteq A_2 + A_2'} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1' \sqsubseteq A_2' \qquad A_1 \sqsubseteq A_2}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \mathsf{inj}'\, t_1' \sqsubseteq \mathsf{inj}'\, t_2' : A_1 + A_1' \sqsubseteq A_2 + A_2'}$$

$$\frac{\begin{array}{c}\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 + A_1' \sqsubseteq A_2 + A_2' \\ \Gamma_1, x_1 : A_1 \sqsubseteq \Gamma_2, x_2 : A_2 \vdash s_1 \sqsubseteq s_2 : B_1 \sqsubseteq B_2 \qquad \Gamma_1, x_1' : A_1' \sqsubseteq \Gamma_2, x_2' : A_2' \vdash s_1' \sqsubseteq s_2' : B_1 \sqsubseteq B_2\end{array}}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \mathsf{case}\, t_1\, \mathsf{of}\ \ \mathsf{inj}\, x_1 : A_1.\, s_1 \mid \mathsf{inj}'\, x_1' : A_1'.\, s_1' \sqsubseteq \mathsf{case}\, t_2\, \mathsf{of}\ \ \mathsf{inj}\, x_2 : A_2.\, s_2 \mid \mathsf{inj}'\, x_2' : A_2'.\, s_2' : B_1 \sqsubseteq B_2}$$

$$\frac{\Gamma_1, x_1 : A_1 \sqsubseteq \Gamma_2, x_2 : A_2 \vdash t_1 \sqsubseteq t_2 : B_1 \sqsubseteq B_2}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \lambda(x_1 : A_1).\, t_1 \sqsubseteq \lambda(x_2 : A_2).\, t_2 : A_1 \to B_1 \sqsubseteq A_2 \to B_2}$$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \to B_1 \sqsubseteq A_2 \to B_2 \qquad \Gamma_1 \sqsubseteq \Gamma_2 \vdash s_1 \sqsubseteq s_2 : A_1 \sqsubseteq A_2}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1\ s_1 \sqsubseteq t_2\ s_2 : B_1 \sqsubseteq B_2}$$

Fig. 18. Syntactic Term Dynamism

$\boxed{\Gamma_1 \sqsubseteq \Gamma_2}$

$$\frac{}{\cdot \sqsubseteq \cdot} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2 \qquad A_1 \sqsubseteq A_2}{\Gamma_1, x_1 : A_1 \sqsubseteq \Gamma_2, x_2 : A_2}$$

Fig. 19. Environment Dynamism

6 GRADUALITY FROM EP PAIRS

We now define and prove graduality of our cast calculus. Graduality, briefly stated, means that if a program is changed to make its types less dynamic, but otherwise the syntax is the same, then the *operational behavior* of the term is "less dynamic"[4] in that either the new term has the same behavior as the old, or it raises a type error, *hiding* some behavior of the original term. Graduality, like parametricity, says that a certain type of syntactic change (making types less dynamic) results in a predictable semantic change (make behavior less dynamic). We define these two notions as *syntactic* and *semantic* term dynamism.

We present syntactic term dynamism in Figure 18, based on the rules of Siek et al. [2015]. Syntactic term dynamism captures the above idea of changing a program to use less dynamic types. If $t_1 \sqsubseteq t_2$, we think of t_2 as being rewritten to t_1 by changing the types to be less dynamic. While we will sometimes abbreviate syntactic term dynamism as $\vdash t_1 \sqsubseteq t_2$, the full form is $\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2$ and is read as "t_1 is syntactically less dynamic than t_2". The syntax evokes the invariant that if you rewrite t_2 to use less dynamic types t_1, then its inputs must be given less dynamic types $\Gamma_1 \sqsubseteq \Gamma_2$ and its outputs must be given less dynamic types $A_1 \sqsubseteq A_2$. We extend type dynamism to environment dynamism in Figure 19 to say $\Gamma_1 \sqsubseteq \Gamma_2$ when Γ_1, Γ_2 have

[4]Here we invoke the meaning of dynamic as "active": less dynamic terms are less active in that they kill the program with a type error where a more dynamic program would have continued to run.

the same length and the corresponding types are related. The rules of syntactic term dynamism capture exactly the idea of "types on the left are less dynamic". Viewed order-theoretically, these rules say that all term constructors are *monotone* in types and terms.

The second piece of graduality is a *semantic formulation* of term dynamism. The intuition described above is that t_1 should be *semantically* less dynamic than t_2 when it has the same behavior as t_2 except possibly when it errors. Note that if $\Gamma_1 = \Gamma_2$ and $A_1 = A_2$, this is exactly what observational error approximation formalizes. Of course, since we can cast between any two types, we can cast any term to be of a different type. Our definition for semantic term dynamism will then be contextual approximation *up to cast*:

Definition 6.1 (Observational Term Dynamism). We say $\Gamma_1 \vdash t_1 : B_1$ is *observationally less dynamic* than $\Gamma_2 \vdash t_2 : B_2$, written $\Gamma_1 \sqsubseteq \Gamma_2 \vDash t_1 \sqsubseteq^{\text{obs}} t_2 : B_1 \sqsubseteq B_2$ when

$$\begin{aligned}\Gamma_1 \vDash \langle B_2 \Leftarrow B_1\rangle t_1 \sqsubseteq^{\text{obs}}\ & \text{let } x_{2,1} = \langle A_{2,1} \Leftarrow A_{1,1}\rangle x_{1,1} \text{ in } \quad : B_2\\ & \vdots \\ & \text{let } x_{2,n} = \langle A_{2,n} \Leftarrow A_{1,n}\rangle x_{1,n} \text{ in}\\ & t_2\end{aligned}$$

where $\Gamma_1 = x_{1,1} : A_{1,1}, \ldots, x_{1,n} : A_{1,n}$ and $\Gamma_2 = x_{2,1} : A_{2,1}, \ldots, x_{2,n} : A_{2,n}$. Or, abbreviated as:

$$\Gamma_1 \vDash \langle B_2 \Leftarrow B_1\rangle t_1 \sqsubseteq^{\text{obs}} \text{let } \Gamma_2 = \langle \Gamma_2 \Leftarrow \Gamma_1\rangle \Gamma_1 \text{ in } t_2 : B_2$$

Note that we have chosen to use the two *upcasts*, but there are three other ways we could have inserted casts to give t_1, t_2 the same type: we can use upcasts or downcasts on the inputs and we can use upcasts or downcasts on the outputs. We will show based on the ep-pair property of upcasts and downcasts that all of these are equivalent (lemma 6.7).

We then define graduality to mean that syntactic term dynamism implies semantic term dynamism:

Theorem 6.2 (Graduality). *If* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2$, *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vDash t_1 \sqsubseteq^{obs} t_2 : A_1 \sqsubseteq A_2$

Proof. By lemma 6.4, theorems 3.7 and 6.9, and corollary 4.8. □

Next, we present our logical relations method for proving graduality. First, to prove an approximation result for terms in λ_G, we will prove approximation for their translations in $\lambda_{T,\mho}$, justified by our adequacy theorem. Second, to prove observational approximation, we will use our logical relation, justified by our soundness theorem. For that we use the following "logical" formulation of term dynamism.

Definition 6.3 (Logical Term Dynamism). For any $[\![\Gamma_1]\!] \vdash t_1 : [\![A_1]\!]$ and $[\![\Gamma_2]\!] \vdash t_2 : [\![A_2]\!]$ with $\Gamma_1 \sqsubseteq \Gamma_2$ and $A_1 \sqsubseteq A_2$, we define $\Gamma_1 \sqsubseteq \Gamma_2 \vDash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2$ as

$$[\![\Gamma_1]\!] \vDash E_{e,A_1,A_2}[t_1] \sqsubseteq \textbf{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \textbf{ in } t_2 : [\![A_2]\!]$$

where the right hand side is defined analogous to the environment cast $\langle \Gamma_2 \Leftarrow \Gamma_1\rangle$.

Lemma 6.4 (Logical Term Dynamism implies Observational Term Dynamism). *For any* $\Gamma_1 \vdash t_1 : A_1$ *and* $\Gamma_2 \vdash t_2 : A_2$ *with* $\Gamma_1 \sqsubseteq \Gamma_2$ *and* $A_1 \sqsubseteq A_2$, *if* $\Gamma_1 \sqsubseteq \Gamma_2 \vDash [\![t_1]\!] \sqsubseteq [\![t_2]\!] : A_1 \sqsubseteq A_2$ *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vDash t_1 \sqsubseteq^{obs} t_2 : A_1 \sqsubseteq A_2$.

Proof. By theorem 4.7 and lemma 4.3. □

Now that we are in the realm of logical approximation, we have all the lemmas of §4.3 at our disposal, and we now start putting them to work. First, as mentioned before, we show that at least with logical term dynamism, the use of upcasts was arbitrary; we could have used downcasts

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \vDash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad A_1 \sqsubseteq B_2 \qquad (A_2 \sqsubseteq B_2 \vee B_2 \sqsubseteq A_2)}{\Gamma_1 \sqsubseteq \Gamma_2 \vDash t_1 \sqsubseteq E_{\langle B_2 \Leftarrow A_2 \rangle}[t_2] : A_1 \sqsubseteq B_2}\ \textsc{Cast-R}$$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \vDash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq A_2 \qquad (A_1 \sqsubseteq B_1 \vee B_1 \sqsubseteq A_1)}{\Gamma_1 \sqsubseteq \Gamma_2 \vDash E_{\langle B_1 \Leftarrow A_1 \rangle}[t_1] \sqsubseteq t_2 : B_1 \sqsubseteq A_2}\ \textsc{Cast-L}$$

Fig. 20. Term Dynamism Upcast, Downcast Rules

instead. The property we need is that the upcast and downcast are *adjoint* (in the language of category theory), also known as a *Galois connection*, which is a basic consequence of the definition of ep pair:

Lemma 6.5 (EP Pairs are Adjoint). *For any ep pair* $(E_e, E_p) : A_1 \triangleleft A_2$, *and terms* $\Gamma \vdash t_1 : A_1, , \Gamma \vdash t_2 : A_2$,

$$\Gamma \vDash E_e[t_1] \sqsubseteq t_2 : A_2 \quad \text{iff} \quad \Gamma \vDash t_1 \sqsubseteq E_p[t_2] : A_1$$

Proof. The two proofs are dual so we show just the $\Rightarrow$ implication. By the retraction property $t_1 \sqsubseteq E_p[E_e t_1]$, so by transitivity it is sufficient to show $E_p[E_e t_1] \sqsubseteq E_p[t_2]$, which follows by congruence and the assumption. □

Lemma 6.6 (Adjointness on Inputs). *If* $\Gamma, x_1 : A_1 \vdash t_1 : B$ *and* $\Gamma, x_2 : A_2 \vdash t_2 : B$, *and* $E_e, E_p : A_1 \triangleleft A_2$, *then*

$$\Gamma, x_1 : A_1 \vDash t_1 \sqsubseteq \mathbf{let}\, x_2 = E_e[x_1]\, \mathbf{in}\, t_2 : B \quad \text{iff} \quad \Gamma, x_2 : A_2 \vDash \mathbf{let}\, x_1 = E_p[x_2]\, \mathbf{in}\, t_1 \sqsubseteq t_2 : B$$

Proof. By a similar argument to lemma 6.5 □

Lemma 6.7 (Alternative Formulations of Logical Term Dynamism). *The following are equivalent*

(1) $[\![\Gamma_1]\!] \vDash E_{e,A_1,A_2}[t_1] \sqsubseteq \mathbf{let}\, [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\, \mathbf{in}\, t_2 : [\![A_2]\!]$
(2) $[\![\Gamma_1]\!] \vDash t_1 \sqsubseteq \mathbf{let}\, [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\, \mathbf{in}\, E_{p,A_1,A_2}[t_2] : [\![A_2]\!]$
(3) $[\![\Gamma_1]\!] \vDash \mathbf{let}\, [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]]\, \mathbf{in}\, t_1 \sqsubseteq E_{p,A_1,A_2}[t_2] : [\![A_2]\!]$
(4) $[\![\Gamma_1]\!] \vDash E_{e,A_1,A_2}[\mathbf{let}\, [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]]\, \mathbf{in}\, t_1] \sqsubseteq t_2 : [\![A_2]\!]$

Proof. By induction on Γ_1, using lemma 6.5 and Lemma 6.6 □

Finally, to prove the graduality theorem, we do an induction over all the cases of syntactic term dynamism. Most important is the cast case $\langle B_1 \Leftarrow A_1 \rangle t_1 \sqsubseteq \langle B_2 \Leftarrow A_2 \rangle t_2$ which is valid when $A_1 \sqsubseteq A_2$ and $B_1 \sqsubseteq B_2$. We break up the proof into 4 atomic steps using the factorization of general casts into an upcast followed by a downcast (lemma 5.15): $E_{\langle A_2 \Leftarrow A_1 \rangle} \sqsupseteq\sqsubseteq E_{p,A_2,?}[E_{e,A_1,?}]$. The four steps are upcast on the left, downcast on the left, upcast on the right, and downcast on the right. These are presented as rules for logical dynamism in Figure 20. Each of the inference rules accounts for two cases. The Cast-R rule says first that if $t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2$ that it is OK to cast t_2 to B_2, as long as B_2 is more dynamic than A_1, and the cast is either an upcast or downcast. Here, our explicit inclusion of $A_1 \sqsubseteq B_2$ in the syntax of the term dynamism judgment should help: the rule says that adding an upcast or downcast to t_2 results in a more dynamic term than t_1, *whenever it is even sensible to ask*: i.e., if it were not the case that $A_1 \sqsubseteq B_2$, the judgment would not be well-formed, so the judgment holds whenever it makes sense! The Cast-L rule is dual.

These four rules, combined with our factorization of casts into upcast followed by downcast suffice to prove the congruence rule for casts (we suppress the context $\Gamma_1 \sqsubseteq \Gamma_2 \vDash$, which is the same in each line):

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[\![t_1]\!] \sqsubseteq [\![t_2]\!] : A_1 \sqsubseteq A_2}{[\![t_1]\!] \sqsubseteq E_{e,A_2,?}[[\![t_2]\!]] : A_1 \sqsubseteq ?}\;\textsc{Cast-R}}{E_{e,A_1,?}[[\![t_1]\!]] \sqsubseteq E_{e,A_2,?}[[\![t_2]\!]] : ? \sqsubseteq ?}\;\textsc{Cast-L}}{E_{p,B_1,?}[E_{e,A_1,?}[[\![t_1]\!]]] \sqsubseteq E_{e,A_2,?}[[\![t_2]\!]] : B_1 \sqsubseteq ?}\;\textsc{Cast-L}}{E_{p,B_1,?}[E_{e,A_1,?}[[\![t_1]\!]]] \sqsubseteq E_{p,B_2,?}[E_{e,A_2,?}[[\![t_2]\!]]] : B_1 \sqsubseteq B_2}\;\textsc{Cast-R}}{[\![\langle B_1 \Leftarrow A_1\rangle t_1]\!] \sqsubseteq [\![\langle B_2 \Leftarrow A_2\rangle t_2]\!] : B_1 \sqsubseteq B_2}\;\textsc{lemma}\ 5.15
$$

Next, we show the 4 rules are valid, as simple consequences of the ep pair property and the decomposition theorem. Also note that while there are technically 4 cases, each comes in a pair where the proofs are exactly dual, so conceptually speaking there are only 2 arguments.

Lemma 6.8 (Upcast, Downcast Dynamism). *The four rules in Figure 20 are valid.*

Proof. In each case we choose which case of lemma 6.7 is simplest.

(1) Cast-L ($A_1 \sqsubseteq B_1 \sqsubseteq A_2$). We need to show $E_{e,B_1,A_2}[E_{e,A_1,B_1}[t_1]] \sqsubseteq \textbf{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \textbf{ in } t_2$. By decomposition and congruence, $E_{e,B_1,A_2}[E_{e,A_1,B_1}[t_1]] \sqsupseteq\sqsubseteq E_{e,A_1,A_2}$ so the conclusion holds by transitivity and the premise.

(2) Cast-R ($A_1 \sqsubseteq B_2 \sqsubseteq A_2$). We need to show $\textbf{let } [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]] \textbf{ in } t_1 \sqsubseteq E_{p,A_1,B_2}[E_{p,B_2,A_2}[t_2]]$. By decomposition and congruence, $E_{p,A_1,B_2}[E_{p,B_2,A_2}[t_2]] \sqsupseteq\sqsubseteq E_{p,A_1,A_2}[t_2]$, so the conclusion holds by transitivity and the premise.

(3) Cast-L ($B_1 \sqsubseteq A_1 \sqsubseteq A_2$). We need to show $E_{p,B_1,A_1}[\textbf{let } [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]] \textbf{ in } t_1] \sqsubseteq E_{p,B_1,A_2}[t_2]$. By decomposition, $E_{p,B_1,A_2}[t_2] \sqsupseteq\sqsubseteq E_{p,B_1,A_1}[E_{p,A_1,A_2}[t_2]]$, so by transitivity it is sufficient to show

$$E_{p,B_1,A_1}[\textbf{let } [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]] \textbf{ in } t_1] \sqsubseteq E_{p,B_1,A_1}[E_{p,A_1,A_2}[t_2]]$$

which follows by congruence and the premise.

(4) Cast-R ($A_1 \sqsubseteq A_2 \sqsubseteq B_2$). We need to show $E_{e,A_1,B_2}[t_1] \sqsubseteq E_{e,A_2,B_2}[\textbf{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \textbf{ in } t_2]$. By decomposition, $E_{e,A_1,B_2}[t_1] \sqsupseteq\sqsubseteq E_{e,A_2,B_2}[E_{e,A_1,A_2}[t_1]]$, so by transitivity it is sufficient to show

$$E_{e,A_2,B_2}[E_{e,A_1,A_2}[t_1]] \sqsubseteq E_{e,A_2,B_2}[\textbf{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \textbf{ in } t_2]$$

which follows by congruence and the premise.

□

The non-cast cases are too long to include here, but are included in the extended version [New and Ahmed 2018]. They are proven using the definitions of the ep pairs for each type connective and the lemmas of §4.3. We note that the proofs are *modular* in that for instance, the proofs about function types only involve the functorial action of the function type and do not depend on any other types being present in the language.

Finally, we prove the graduality theorem by induction on syntactic term dynamism derivations, finishing the proof of theorem 6.2.

Theorem 6.9 (Logical Graduality).
If $\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1$, *then* $\Gamma_1 \sqsubseteq \Gamma_2 \vDash [\![t_1]\!] \sqsubseteq [\![t_2]\!] : A_1 \sqsubseteq A_2$.

7 RELATED WORK AND DISCUSSION

Our analysis of graduality as observational approximation and dynamism as ep pairs builds on the axiomatic and denotational semantics of graduality for a call-by-name language presented in [New and Licata 2018]. The semantics there gives axioms of type and term dynamism that imply that upcasts and downcasts are embedding-projection pairs. Our analysis here is complementary: we present the graduality theorem as a concrete property of a gradual language defined with an operational semantics. Our graduality logical relation should serve as a concrete *model* of a call-by-value version of gradual type theory, similar to the call-by-name denotational models presented there. Furthermore, we show here how this interpretation of graduality maps back to a standard cast calculus presentation of gradual typing.

Graduality vs Gradual Guarantee. The notion of graduality we present here is based on the *dynamic gradual guarantee* by Siek, Vitousek, Cimini, and Boyland [Siek et al. 2015; Boyland 2014]. The dynamic gradual guarantee says that syntactic term dynamism is an *invariant* of the operational semantics up to error on the less dynamic side. More precisely, if $\cdot \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2$ then either $t_1 \mapsto^* \mho$ or both t_1, t_2 diverge or $t_1 \mapsto^* v_1$ and $t_2 \mapsto^* v_2$ with $v_1 \sqsubseteq v_2$. Observe that when restricting $A_1 = A_2 = 1$, this is precisely the relation on closed programs out of which we build our definition of semantic term dynamism. We view their formulation of the dynamic gradual guarantee as a syntactic *proof technique* for proving graduality of the system.

Graduality should be easier to formulate for different presentations of gradual typing because it does not require a second syntactic notion of term dynamism for the implementation language. In the proofs of the gradual guarantee in Siek et al. [2015], they have to develop new rules for term dynamism for their cast calculus, that they do not attempt to justify at an intuitive level. Additionally, they have to change their translation from the gradual surface language to the cast calculus, because the traditional translation did not preserve the rigid syntactic formulation of term dynamism. In more detail, when a dynamically typed term $t : ?$ was applied to a term $s : A$, in their original formulation this was translated as

$$[\![t\ s]\!] = (\langle (A \to ?) \Leftarrow ? \rangle [\![t]\!])\ [\![s]\!]$$

but if the term in function position had a function type $t' : ? \to ?$, it was translated as

$$[\![t'\ s]\!] = ([\![t]\!]\ (\langle ? \Leftarrow A \rangle [\![s]\!])$$

But if $t' \sqsubseteq t$, we would not have $[\![t's]\!] \sqsubseteq [\![ts]\!]$ because the function position on the left has type $? \to ?$ which is *more dynamic* than on the right which has $A \to ?$. While changing this was perfectly reasonable to do to use their syntactic proof method, we can see that from the *semantic* point of view of graduality there was nothing wrong with their original translation and it could have been validated using a logical relation.

Another significant difference between our work and theirs is that we identify the central role of embedding-projection pairs in graduality, and take advantage of it in our proof. As mentioned above, they add rules to term dynamism for the cast calculus without justification. These rules are the generalization of our Cast-R and Cast-L *without* the restriction that the casts be upcasts or downcasts:

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad A_1 \sqsubseteq B_2}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq \langle B_2 \Leftarrow A_2 \rangle t_2 : A_1 \sqsubseteq B_2}\ \textsc{Cast-R'} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq A_2}{\Gamma_1 \sqsubseteq \Gamma_2 \vdash \langle B_1 \Leftarrow A_1 \rangle t_1 \sqsubseteq t_2 : B_1 \sqsubseteq A_2}\ \textsc{Cast-L'}$$

These are valid rules in our system, but by identifying the subset of upcasts and downcasts, we prove the validity of the rules from earlier, intuitive rules: decomposition, congruence, and the ep-pair properties. Furthermore, while we do not take these rules as primitive it is notable that these two rules imply that upcasts and downcasts are adjoint—i.e., if $A_1 \sqsubseteq A_2$, the following are provable for $t : A_1$ and $s : A_2$:

$$t \sqsubseteq \langle A_1 \Leftarrow A_2\rangle\langle A_2 \Leftarrow A_1\rangle t \qquad \langle A_2 \Leftarrow A_1\rangle\langle A_1 \Leftarrow A_2\rangle s \sqsubseteq s$$

Siek et al. [2015] also present a theorem called the *static* gradual guarantee that pertains to the type checking of gradually typed programs. The static gradual guarantee says that if a term $\Gamma \vdash t_1 : A_1$ type checks, and t_2 is syntactically more dynamic, then $\Gamma \vdash t_2 : A_2$ with a more dynamic type, i.e., $A_1 \sqsubseteq A_2$. We view this as a *corollary* to graduality. If type checking is a compositional procedure that seeks to rule out dynamic type errors, then if t_1 is syntactically less dynamic than t_2, then it is also semantically less dynamic, meaning every type error in t_2's behavior was already present in t_1, so it should also type check.

Types as EP Pairs. The interpretation of types as retracts of a single domain originated in Scott [1972] and is a common tool in denotational semantics, especially in the presence of a convenient *universal* domain. A retraction is a pair of morphisms $s : A \to B$, $r : B \to A$ that satisfy the retraction property $r \circ s = \mathrm{id}_A$, but not necessarily the projection property $s \circ r \sqsubseteq_{\mathrm{err}} \mathrm{id}_B$. Thus ep pair semantics can be seen as a more refined retraction semantics. Retractions have been used to study interaction between typed and untyped languages, e.g., see Benton [2005]; (Favonia) et al. [2017].

Embedding-projection pairs are used extensively in domain theory as a technical device for solving non-well-founded domain equations, such as the semantics of a dynamic type. In this paper, our error-approximation ep pairs do not play this role, and instead the retraction and projection properties are desirable in their own right for their intuitive meaning for type checking.

Many of the properties of our embedding-projection pairs are anticipated in Henglein [1994] and Thatte [1990]. Henglein [1994] defines a language with a notion of *coercion* $A \rightsquigarrow B$ that corresponds to general casts, with primitives of tagging $tc! : tc(?, \ldots) \rightsquigarrow ?$ and untagging $tc? : ? \rightsquigarrow tc(?, \ldots)$ for every type constructor "tc". Crucially, Henglein notes that $tc!; tc?$ is the identity modulo efficiency and that $tc?; tc!$ errors more than the identity. Furthermore, they define classes of "positive" and "negative" coercions that correspond to embeddings and projections, respectively, and a "subtyping" relation that is the same as type precision. They then prove several theorems analogous to our results:

(1) (Retraction) For any pair of positive coercion $p : A \rightsquigarrow B$, and negative coercion $n : B \rightsquigarrow A$, they show that $p; n$ is equal to the identity in their equational theory.
(2) (Almost projection) Dually, they show that $n; p$ is equal to the identity *assuming* that $tc?; tc!$ is equal to the identity for every type constructor.
(3) They show every coercion factors as a positive cast to ? followed by a negative cast to ?.
(4) They show that $A \leq B$ if and only if there exists a positive coercion $A \rightsquigarrow B$ and a negative coercion $B \rightsquigarrow A$.

They also prove factorization results that are similar to our factorization definition of semantic type precision, but it is unclear if their theorem is stronger or weaker than ours. One major difference is that their work is based on an equational theory of casts, whereas ours is based on notions of observational equivalence and approximation of a standard call-by-value language. Furthermore, in defining our notion of observational error approximation, we provide a more refined projection property, justifying their use of the term "safer" to compare $p; e$ and the identity.

The system presented in Thatte [1990], called "quasi-static typing" is a precursor to gradual typing that inserts type annotations into dynamically typed programs to make type checking explicit. There they prove a desirable soundness theorem that says their type insertion algorithm produces an explicitly coercing term that is minimal in that it errors no more than the original dynamic term. They prove this minimality theorem with respect to a partial order $\sqsupseteq$ defined as a logical relation over a domain-theoretic semantics that (for the types they defined) is analogous to our error ordering for the operational semantics. However, they do not define our operational formulation of the ordering as contextual approximation, linked to the denotational definition by the adequacy result, nor that any casts form embedding-projection pairs with respect to this ordering.

Finally, we note that neither of these papers [Henglein 1994; Thatte 1990] extends the analysis to anything like graduality.

Semantics of Casts. Superficially similar to the embedding-projection pair semantics are the *threesome casts* of Siek and Wadler [2010]. A threesome cast factorizes an arbitrary cast $A \Rightarrow B$ through a third type C as a *downcast* $A \Rightarrow C$ followed by an *upcast* $C \Rightarrow B$, whereas ep-pair semantics factorizes a cast as an *upcast* $A \Rightarrow ?$ followed by a *downcast* $? \Rightarrow B$. Threesome casts can be used to implement gradual typing in a space-efficient manner, the third type C is used to collapse a sequence of arbitrarily many casts into just the two. In the general case, the threesome cast $A \Rightarrow C \Rightarrow B$ is *stronger* (fails more) than the direct cast $A \Rightarrow B$. This is the point of threesome casts: the middle type faithfully represents a sequence of casts in minimal space. EP pair semantics instead factorizes a cast $A \Rightarrow B$ into an *upcast* $A \Rightarrow ?$ followed by a *downcast* $? \Rightarrow B$, a factorization already utilized in [Henglein 1994], and which we showed is *always* equivalent to the direct cast $A \Rightarrow B$. We view the benefits of the techniques as orthogonal: the up-down factorization helps to prove graduality, whereas the down-up factorization helps implementation. The fact that both techniques reduce reasoning about arbitrary casts to just upcasts and downcasts supports the idea that upcasts and downcasts are a fundamental aspect of gradual typing.

Recently, work on *dependent interoperability* [Dagand et al. 2016, 2018] has identified Galois connections as a semantic formulation for casting between more and less precise types in a non-gradual dependently typed language, and conjectures that this should relate to type dynamism. We confirm their conjecture in showing that the casts in gradual typing satisfy the slightly stronger property of being embedding-projection pairs and have used it to explain the cast semantics of gradual typing and graduality. Furthermore, our analysis of the precision rules as compositional constructions on ep pairs is directly analogous to their library, which implements "connections" between, for instance, function types given connections between the domains and codomains using Coq's typeclass mechanism.

Pairs of Projections and Blame. One of the main inspirations for this work is the analysis of contracts in Findler and Blume [2006]. They decompose contracts in untyped languages as a pair of "projections", i.e., functions $c : ? \to ?$ satisfying $c \sqsubseteq_{? \to ?} \mathrm{id}$. However, they do not provide a rigorous definition or means to prove this ordering for complex programs as we have. There is a close relationship between such projections and ep pairs (an instance of the relationship between adjunctions and (co)monads): for any ep pair $e, p : A \triangleleft B$, $e \circ p : B \to B$ is a projection. However, we think this relationship is a red herring: instead we think that a pair of projections is better understood as ep pairs themselves. The intuition they present is that one of the projections restricts the behavior of the "positive" party (the term) and the other restricts the behavior of the "negative" party (the continuation). EP pairs are similar, the projection restricts the positive party by directly checking, and the embedding restricts the negative party in the function case by calling a projection on any value received from its continuation. However, in our current formulation, it does not even make sense to ask if each component of our embedding-projection pairs is a projection because the definition of a projection assumes that the domain and codomain are the same (to define the composite $c \circ c$). We conjecture that this can be made sensible by using a PER semantics where types are relations on untyped values, so that the embedding and projection have “underlying” untyped terms representing them, and those are projections.

Their analysis of blame was adapted to gradual typing in Wadler and Findler [2009] and plays a complementary role to our analysis: they use the dynamism relation to help prove the blame soundness theorem, whereas we use it to prove graduality. The fact that they use essentially the same solution suggests there is a deeper connection between blame and graduality than is currently understood.

Gradualization. The Gradualizer [Cimini and Siek 2016, 2017] and Abstracting Gradual Typing (AGT) [Garcia et al. 2016b] both seek to make language design for gradually typed languages more systematic. In doing so they make proving graduality far easier than our proof technique possibly could: it holds by construction. Furthermore, these systems also provide a surface-level syntax for gradual typing and an explanation for gradual type checking, while we do not address these at all. However, the downside of their approaches is that they require a rigid adherence to a predefined language framework. While our gradual cast calculus as presented fits into this framework, many gradually typed languages do not. For instance, Typed Racket, the first gradually typed language ever implemented [Tobin-Hochstadt and Felleisen 2008], is not given an operational semantics in the style of a cast calculus, but rather is given a semantics *by translation* to an untyped language using contracts. We could prove the graduality of such a system by adapting our logical relation to an untyped setting.

We hope in the future to explore the connections between the above frameworks and our analysis of dynamism as embedding-projection pairs. We conjecture that both Gradualizer and AGT by construction produce upcasts and downcasts that satisfy the ep pair properties. The AGT approach in particular has some similarities that stand out: their formulation of type dynamism is based on an embedding-projection pair between static types and sets of gradual types. However, we are not sure if this is a coincidence or has a deeper connection to our approach.

8 CONCLUSION

Graduality is a key property for gradually typed languages as it validates programmer intuition that adding precise types only results in stricter type checking. Graduality is challenging to prove. Moreover, it rests upon the language’s definition of type dynamism but there has been little guidance on defining type dynamism, other than that graduality must hold. We have given a semantics for type dynamism: $A \sqsubseteq B$ should hold when the casts between A, B form an embedding-projection pair. This allows for natural proofs of graduality using a logical relation for observational error approximation.

Looking to the future, we would like to make use of our semantic formulation of type dynamism based on ep pairs to design and analyze gradual languages with advanced features such as parametric polymorphism, effect tracking, and mutable state. For parametric polymorphism in particular, we would like to investigate whether our approach justifies any of the type-dynamism definitions previously proposed [Ahmed et al. 2017; Igarashi et al. 2017a], and the possibility of proving both graduality and parametricity theorems with a single logical relation.

ACKNOWLEDGMENTS

We gratefully acknowledge the valuable feedback provided by Ben Greenman and the anonymous reviewers. Part of this work was done at Inria Paris while Amal Ahmed was a Visiting Professor.

This material is based upon work supported by the National Science Foundation under grant CCF-1453796, and the European Research Council under ERC Starting Grant SECOMP (715753).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our funding agencies.

REFERENCES

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

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.*

Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. 2013. Gradual Typing for Smalltalk. *Science of Computer Programming* (Aug. 2013). Available online.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. 283–295.

Nick Benton. 2005. Embedded Interpreters. *Journal of Functional Programming* 15, 04 (2005), 503–542.

John Tang Boyland. 2014. The Problem of Structural Type Tests in a Gradual-Typed Language. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings.*

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-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. 2016. Partial Type Equivalences for Verified Dependent Interoperability *(ICFP 2016)*. 298–310.

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

Keun-Bang Hou (Favonia), Nick Benton, and Robert Harper. 2017. Correctness of compiling polymorphism to dynamic typing. *Journal of Functional Programming* 27 (2017).

Matthias Felleisen and Robert Hieb. 1992. A Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (1992), 235–271.

Robby Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *International Symposium on Functional and Logic Programming (FLOPS).*

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

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

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016b. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).*

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: Hybrid Checking for Flexible Specifications. In *Scheme and Functional Programming Workshop (Scheme)*. 93–104.

Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming* 22, 3 (1994), 197–230.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient Gradual Typing. *Higher Order Symbol. Comput.* 23, 2 (June 2010).

Atsushi Igarashi, Peter Thiemann, Vasco Vasconcelos, and Philip Wadler. 2017b. Gradual Session Types. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom.*

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

Lintaro Ina and Atsushi Igarashi. 2011. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11).*

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. 775–788.

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs (Extended Version). *ArXiv e-prints* (July 2018). arXiv:1807.02786

Max S. New and Daniel R. Licata. 2018. Call-by-Name Gradual Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Hélène Kirchner (Ed.), Vol. 108. http://drops.dagstuhl.de/opus/volltexte/2018/9194

Dana Scott. 1972. Continuous lattices. In *Toposes, algebraic geometry and logic*. 97–136.

Jeremy Siek, Micahel Vitousek, Matteo Cimini, and John Tang Boyland. 2015. 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 and Philip Wadler. 2010. Threesomes, with and without blame. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain.* 365–376.

Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. 2014. Gradual typing embedded securely in JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California.* 425–438.

Satish Thatte. 1990. Quasi-static typing. In *ACM Symposium on Principles of Programming Languages (POPL).* 367–381.

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

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California.*

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

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual Typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11).*