Semantic Foundations for Gradual Typing (Extended Draft)

MAX NEW, Institution1
AMAL AHMED, Institution1

Gradually typed languages allow statically typed and dynamically typed code to interact while maintaining benefits of both styles. With such an enticing goal, gradual typing has eagerly been adapted for many different languages and language features. However, most work has been ad hoc, with little to build upon except for analogies with syntactic type checking.

We propose that *embedding-projection pairs* (ep pairs) provide a *semantic* foundation to aid in the design and analysis of gradually typed languages. We construct an ep pair semantics that decomposes complex cast calculi into simple-to-understand pieces, allowing us to abstract over common patterns of cast construction and easily extend the language with new types. This gives a new perspective on existing gradual type systems. In particular, *type precision* plays a central role, and we give a definition *semantic type precision*, analogous to semantic subtyping. In particular, we introduce a new form of gradual refinement types that allow a programmer to define new types by implementing embedding-projection pairs.

Finally, we show that the ep pair semantics enables a straightforward metatheory for gradual typing. We define a denotational semantics to make precise the relationship with the classical use of ep pairs in domain theory. Additionally, we develop a formulation of the gradual guarantee as a kind of contextual approximation, and prove it using a novel logical relation directly defined from the ep pair semantics.

CCS Concepts: •**Software and its engineering** → **General programming languages;** •**Social and professional topics** → *History of programming languages;*

ACM Reference format:
Max New and Amal Ahmed. 2017. Semantic Foundations for Gradual Typing (Extended Draft). *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 57 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Gradually[1] typed programming languages are intended to resolve the conflict between static and dynamically typed programming styles. A gradual language should support adding types to dynamically typed programs and safe interoperation between more precisely typed and less precisely typed components. With this promising ideal, many different language features have been "gradualized" including refinement type, objects, sessions, effects and polymorphism [1, 2, 15, 16, 19, 26].

The proliferation of work on gradual typing has led many to wonder if the gradualization process can be more systematic, even automatic. In [24], the authors propose design criteria that a gradually typed language should satisfy. Notably, they define the "gradual guarantee" as a central soundness theorem. Two approaches, the Gradualizer [4, 5] and Abstracting Gradual Typing [13] go further and seek to fully automate the "gradualization" process in a way that automatically satisfies the gradual guarantee. Both start with the syntax of an existing statically typed language and add a dynamic type and a *cast calculus*, introducing runtime type errors to the semantics. However, there is a seemingly quite different approach to gradual typing taken by languages like Typed Racket

[1]In this paper we use blue to typeset our surface language and red to typeset the core language. The paper will be much easier to read if viewed/printed in color.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

that these methods do not capture. Typed Racket [30, 31] is defined by starting instead with an existing *dynamically* typed language Racket which already has errors, and implementing checking using a contract system, meaning that runtime type checking is done by ordinary dynamically typed programs. To fit these cast calculi and contract systems into one conceptual framework, we propose a semantic understanding of gradual types as *embedding-projection pairs* (ep pairs) that naturally produces languages that satisfy the gradual guarantee. Since the similarity to contract systems is more obvious (see §8), we focus mostly on the relationship to languages based on cast calculi.

Sound Gradual Typing Necessitates EP Pairs. We argue that embedding-projection pairs are necessarily present in all sound higher-order gradually typed languages, and motivate the definition of ep pairs with an informal description. For a language to be *gradual*, it should allow linking any typed code with programs of the dynamic type. On the other hand, for a gradually typed language to be *sound*, types should enable extensional reasoning. So for example, functions at the type $\mathbb{N} \to \mathbb{N}$ should be equal when they take equal numbers to equal numbers, even if they were cast from functions $? \to ?$ that differed when called on a boolean value `true`. To achieve soundness, the process of linking typed and untyped code must insert casts that ensure that the components only interact in the way dictated by the type. In a higher-order language, this means that any type A must come equipped with an upcast (embedding) $e : A \to ?$ to the dynamic type and a downcast (projection) $p : ? \to A$ from the dynamic type that possibly errors. In order to reason about these mixed-precision programs, we would expect that a precisely typed value, when passed (upcast) to a dynamically typed component and eventually returned back (downcast) should be equivalent to its original value, since it should satisfy whatever the cast is checking. This is the *retraction* property: $p(e(x)) \cong x$. On the other hand, when a dynamically typed value is passed to a statically typed portion of the program and back, it is not necessarily equal to its original value, because in order to ensure it behaved as a typed value, dynamic type errors might be thrown. However, where the value did behave correctly, it should be equal. This is known as the *projection* property: $e(p(x)) \sqsubseteq_{\mathrm{err}} x$, which says that casting to a more precise type and back should only add errors to a value. These two properties, retraction and projection, define what it means for e, p to be an embedding-projection pair from A to ?, which we denote $(e, p) : A \triangleleft ?$, and so *the casts to and from the dynamic type in any sound gradually typed language should form an ep pair.*

We advocate going further, not only are ep pairs present in gradually typed languages, they are the natural semantic primitives on which gradual typing is based. For instance, we can decompose the definitions of all casts $A \Rightarrow B$ as an embedding to the dynamic type $A \Rightarrow ?$ followed by a projection $? \Rightarrow B$, and these ep pairs can be given a simple, compositional definition. This allows us to abstract over commonalities between casts at different types in our semantics and logical relation, and also helps to understand the structure of more complex types like the gradual record types of [13]. Our semantics also enables us to define a *semantic* notion of type precision for which existing syntactic rules can be proved sound.

Since the notion of ep pair can be made precise in any language with errors, ep pair semantics provide a means to *implement* and *understand* gradual typing by translation to a non-gradually-typed language. We take this approach, reducing soundness theorems for our gradually typed surface language to reasoning about programs in a standard core language with errors and a recursive type (§4). Furthermore, this gives a framework to guide the design of new gradually typed languages by studying what ep pairs can be defined using different language features.

The modularity of ep pair semantics makes the system inherently extensible, meaning we can enable expert users to extend the language by programming new ep pairs directly in the surface language, a feature we call *cast refinement types* (§7). These bring the natural extensibility of contract systems to gradually typed languages, and our semantics shows that if all user-defined types use valid ep pairs, the language as a whole will maintain the gradual guarantee.

Denotational Semantics. The idea that there is a connection between ep pairs and dynamic typing is certainly not new. In fact, ep pairs are central to the original denotational semantics of untyped lambda calculus [22], and domain-theoretic semantics of recursive types more generally [28]. However, the classical denotational semantics identifies non-termination and type errors, so a standard semantics is not adequate for proving a property like the gradual guarantee. We adapt this old technique to the gradually typed setting by developing a suitable domain theory where parallel to the usual domain order $\leq_{\text{div}}$ is an "error ordering" $\sqsubseteq_{\text{err}}$ which has a type error, rather than a diverging program, as a least element. We use this domain theory to give an adequate denotational semantics for our language and prove soundness of our ep pair semantics.

Gradual Guarantee. The gradual guarantee was proposed in [24] as a design criterion for gradually typed languages. Informally, it states that making the types of a gradually typed program more precise, i.e. replacing the dynamic type ? with a more specific type like `bool`, should only add runtime type checking (i.e., possibly type errors) to the program, and not otherwise change behavior. In that paper, they prove the gradual guarantee by an operational simulation argument whose details are quite tied to the specific cast calculus used. We show that the ep pair semantics helps to understand and prove the gradual guarantee. First, we see that the notion of "only adding errors" is precisely the error ordering we needed to precisely define ep pairs in the first place. Then we extend the gradual guarantee to open terms as a form of *heterogeneous contextual approximation*. Finally, we show how the ep pairs define a notion of *cross-type error ordering* that can be used to define a logical relation sound for the open gradual guarantee. With this view, the type precision judgment is a formal syntax for logical relatedness arguments, ala [21]. By decomposing our types and ep pairs, we simplify the construction of the logical relation and in particular provide a very simple soundness proof.

For reasons of space, most proofs have been elided, but more details can be found in the supplementary material.

2 DECONSTRUCTING GRADUAL TYPING

We present our semantics of gradual typing in three stages: a gradually typed surface language, an elaboration of the surface into a non-gradually typed core language, and a domain-theoretic denotational semantics for the core language. In this section, we give a high-level view of the semantics and how ep pairs decompose constructions into simpler pieces. In contrast, most semantics for gradually typed languages are defined by a *cast calculus* where the fundamental unit is the cast $\mathsf{A} \Rightarrow \mathsf{B}$. However, casts have few closure properties and their dynamic semantics are not defined in a compositional way. For instance, the composition of two casts $\mathsf{A} \Rightarrow \mathsf{B} \Rightarrow \mathsf{C}$ generally checks a much stronger property than the direct cast $\mathsf{A} \Rightarrow \mathsf{C}$. For example $\mathbb{B} \Rightarrow \mathsf{A} \to \mathsf{B} \Rightarrow \mathbb{B}$ will always fail in most languages, whereas the direct cast $\mathbb{B} \Rightarrow \mathbb{B}$ is the identity function. Instead of taking *arbitrary* casts as fundamental, we take the subset of casts that are components of embedding-projection pairs. Embedding-projection pairs admit many constructions, for example the composition of $e, p : A \triangleleft B$ with $e', p' : B \triangleleft C$ is an ep pair from A to C with embedding $e' \circ e$ and projection $p \circ p'$. Furthermore, nothing is lost by focusing on the ep pairs because the semantics of all casts can be recovered, by decomposing $\mathsf{A} \Rightarrow \mathsf{B}$ into the composition $\mathsf{A} \Rightarrow ? \Rightarrow \mathsf{B}$[2]. While not the most efficient *implementation*, this can serve as a simple semantic specification for a cast calculus.

[2]Note that this is not the same as threesome casts, which are a downcast followed by an upcast.

Furthermore the definition of these embeddings $A \Rightarrow ?$ and projections $? \Rightarrow A$ can be given a definition by induction on A. We start with the well-known function contract $A \to B \Rightarrow ?$ constructs a proxied function that projects A and embeds B. In an untyped language this might be implemented as:$\lambda f.\lambda x.e_B(f(p_A x))$ But when interpreted in a typical *typed* language, this syntax would be defining a cast $A \to B \Rightarrow ? \to ?$, the untyped syntax obscures the implicit operation of tagging the function at the dynamic type. So then to define the correct embedding, we need a tagging operation $e_{? \to ?} : ? \to ? \Rightarrow ?$. When, as in most untyped functional languages, the dynamic type is a disjoint sum, this will just be the injection. Then the correct embedding and projection are

$$\begin{aligned} e_{A \to B} &= \lambda f : A \to B.e_{? \to ?}(\lambda x : ?.e_B(f(p_A x))) \\ p_{A \to B} &= \lambda d : ?.\,\mathtt{let}\, f = p_{? \to ?} d \,\mathtt{in}\, \lambda x : A.p_B(f(e_A(x))) \end{aligned}$$

where for a sum-like dynamic type $p_{? \to ?}$ does a case analysis, erroring if the value is not a function.

Constructive Type Precision. The above shows that when defining a cast $A \to B \Rightarrow ?$, the definition naturally uses casts like $A \to B \Rightarrow ? \to ?$ that do not have $?$ on the right. So for our syntax to more closely match our semantics, we see that our definition of embedding-projection pairs should not just be defined by recursion on *types* but some more general syntax for ep pairs $A \triangleleft B$. Fortunately, previous gradual typing work has introduced an appropriate syntax, without even knowing it: the syntax of *type precision derivations*! That is, we can give a *constructive* interpretation of type precision, defining by recursion on the *derivation* $d :: A \sqsubseteq B$ an ep pair from A^+ to B^+. Type precision[3] was introduced to capture the notion of one type being "less dynamic" than another. That is, when $A \sqsubseteq B$, then A is a less dynamic or "more static" type than B, so the type checker is stricter when A is used than when B is used. For example, $?$ is the most imprecise type $A \sqsubseteq ?$ and so the type checker is very lax when $?$ is used, enabling a dynamically typed programming style. We give a more semantic intuition: $A \sqsubseteq B$ means that casting from $A \Rightarrow B$ loses no information, that is, the casts $A \Rightarrow B, B \Rightarrow A$ form an ep pair! In fact, a precision derivations $A \sqsubseteq B$ *defines* the casts $A \Rightarrow B, B \Rightarrow A$. Our semantics of casts proceeds in two stages, first we define for any type precision derivation $d :: A \sqsubseteq B$, ep pairs $[\![d^e]\!] : A^+ \to B^+, [\![d^p]\!] : B^+ \to A^+$. Then we define for every type A, a distinguished precision derivation that we call the "imprecise derivation" and denote $\mathrm{Imp}(A)$ showing that $?$ is the most imprecise type. This derivation $\mathrm{Imp}(A) :: A \sqsubseteq ?$ is used with the previous semantics to define the ep pairs $[\![A^e]\!] : A \to ?, [\![A^p]\!] : ? \to A$. The construction of this derivation then reveals the structure underlying our casts. For example, the derivation for the function ep pair is decomposed explicitly as described above into an ep pair to the type of dynamic functions $? \to ?$ and then tagged at the dynamic type:

$$\dfrac{\dfrac{d_A :: A \sqsubseteq ? \qquad d_B :: B \sqsubseteq ?}{d_A \to d_B :: A \to B \sqsubseteq ? \to ?} \qquad \dfrac{}{\mathrm{Tag}_{? \to ?} :: ? \to ? \sqsubseteq ?}}{d_A \to d_B;\mathrm{Tag}_{? \to ?} :: A \to B \sqsubseteq ?}$$

which describes precisely the definition of the ep pair for the function type. For other types, such as sums and products, the ep pairs are constructed the same way, by composing a natural lift of ep pairs with a primitive tagging/untagging pair. We take advantage of this commonality throughout our analysis to simplify certain semantic arguments.

Gradual Records. As a more complex example of decomposing a cast, we consider the gradual record types of [13], that provide a simple model of gradual objects. These records come in two flavors, the *precise* record type $\{\eta_1 : A_1, \ldots, \eta_n : A_n\}$ which is an ordinary named tuple type, and the *gradual* record type $\{\eta_1 : A_1, \ldots, \eta_n : A_n, ?\}$ (notice the $?$), which in addition to supporting the fields $\eta_1 : A_1, \ldots, \eta_n : A_n$, may or may not contain other fields in some fixed universe of field names.

[3]Sometimes naïvely referred to as naïve subtyping due to the covariant function rule.

These record types have an interesting precision structure. Both admit a depth precision rule:

$$\frac{\forall i \in 1,\dots,n.\ A_i \sqsubseteq B_i}{\{\eta_1 : A_1,\dots,\eta_n : A_n\} \sqsubseteq \{\eta_1 : B_1,\dots,\eta_n : B_n\}} \qquad \frac{\forall i \in 1,\dots,n.\ A_i \sqsubseteq B_i}{\{\eta_1 : A_1,\dots,\eta_n : A_n, ?\} \sqsubseteq \{\eta_1 : B_1,\dots,\eta_n : B_n, ?\}}$$

but only gradual record types satisfy a *width* precision rule, which we restrict to $?$ for semantic reasons explained below, noting that arbitrary width precision can be recovered using composition/transitivity:

$$\{\eta_1 : A_1,\dots,\eta_n : A_n, \eta_{n+1} : ?,\dots,\eta_{n+m} : ?, ?\} \sqsubseteq \{\eta_1 : A_1,\dots,\eta_n : A_n, ?\}$$

And finally, a precise record type is more precise than the gradual record with the same fields:

$$\{\eta_1 : A_1,\dots,\eta_n : A_n\} \sqsubseteq \{\eta_1 : A_1,\dots,\eta_n : A_n, ?\}$$

We can understand these rules by deconstructing the record types into simpler pieces. The precise record type is just a named tuple, and our elaboration process reflects this:

$$\{\eta_1 : A_1,\dots,\eta_n : A_n\}^+ = \bigtimes_{i\in\{1,\dots,n\}} {A_{\eta_i}}^+$$

The gradual records, on the other hand, are implemented as a pair of a precise record and a partial product of dynamically typed values:

$$\{\eta_1 : A_1,\dots,\eta_n : A_n, ?\}^+ = (\bigtimes_{i\in\{1,\dots,n\}} {A_{\eta_i}}^+) \times (\prod_{\mathfrak{F}-\{\eta_1,\dots,\eta_n\}} ?)$$

where $\mathfrak{F}$ is the set of all field names. Here $\prod_{\mathfrak{F}-\{\eta_1,\dots,\eta_n\}} ?$ is essentially a partial function from $\mathfrak{F} - \{\eta_1,\dots,\eta_n\}$ to $?$, meaning when projecting a field, either a dynamically typed value is returned or an error is raised.

Then we can justify the precision rules (or lack theoreof) by this semantics. The ep pair for the depth rules is easily constructed by applying the embeddings/projections to each field. For the precise-to-gradual rule, the projection just projects out the precise side, and the embedding uses the partial function that always errors. This satisfies the projection property because the always erroring function is the $\sqsubseteq_{\text{err}}$-least element of $\prod_{\mathfrak{F}-\{\eta_1,\dots,\eta_n\}} ?$. For the gradual width rule, the embedding adds the values in the extra fields $\eta_{n+1} : A_{n+1},\dots,\eta_{n+m} : A_{n+m}$ into the partial function. The projection projects each of the fields out of the partial function component, erroring if the fields are undefined. Now we also see why we restricted the width rule to forget fields that are at the dynamic type: otherwise this rule would build in the ep pair for arbitrary other types. Finally, we can see why width precision is *not* valid for precise records, because while we could define a possible embedding by forgetting the extra fields, the only possible projection would always error, which would not satisfy the retraction property.

We can then piece together these precision rules to define casts for a precise record type. First, you use depth precision to make every field dynamically typed. Then embed as a gradual record type and use width precision to remove all precise fields. Finally we use the tag on the dynamic type, this time a tag for the maximally imprecise gradual record $\{?\}$:

$$\frac{\{\eta_1 : A_1,\dots,\eta_n : A_n\} \sqsubseteq \{\eta_1 : ?,\dots,\eta_n : ?\} \sqsubseteq \{\eta_1 : ?,\dots,\eta_n : ?, ?\} \sqsubseteq \{?\} \sqsubseteq ?}{\{\eta_1 : A_1,\dots,\eta_n : A_n\} \sqsubseteq ?}$$

Semantic Type Precision. Reading type precision derivations as ep pairs doesn't fully explain type precision. One issue is that type precision has always been interpreted as a *proof-irrelevant proposition*, either $A \sqsubseteq B$ holds or it doesn't, but there can be many different ep pairs between the *same* two types, a fact we exploit in §7. For instance, there are many ep pairs from $\mathbb{N}$ to $\mathbb{N}$, such as the identity, but also the embedding that multiplies by 2, which encodes even numbers. Then we have a natural question of *coherence* of our semantics for type precision derivations: we want that any derivations $d, d' :: A \sqsubseteq B$ of the same precision judgment represent the same ep pair:

$\mathsf{d}^{\mathsf{e}} = \mathsf{d}'^{\mathsf{e}}$ and $\mathsf{d}^{\mathsf{p}} = \mathsf{d}'^{\mathsf{p}}$. In fact, we will do one better than just coherence: we show that the ep pair represented by a derivation of type precision satisfies a property that uniquely characterizes it, we call this property *semantic type precision*.

To understand this property, we recall some *syntactic* intuitions for type precision. In [24], the *static gradual guarantee* states that if $\mathsf{A} \sqsubseteq \mathsf{B}$ then if a program t type checks with a type annotation of A, changing that annotation to B should still type check. Semantically, changing the type annotation means replacing casts $\mathsf{A} \Rightarrow \mathsf{C}$ or $\mathsf{C} \Rightarrow \mathsf{A}$, with casts $\mathsf{B} \Rightarrow \mathsf{C}$, $\mathsf{C} \Rightarrow \mathsf{B}$. If the type checker tries to rule out runtime type errors, this means that if $\mathsf{A} \sqsubseteq \mathsf{B}$, a program with casts involving A should *error more* than a program with casts involving B, meaning that A encodes a stricter interface than B. Intuitively this must be true if any time we have a cast $\mathsf{C} \Rightarrow \mathsf{A}$, the cast $\mathsf{C} \Rightarrow \mathsf{B}$ is *already being run*, which is true if we can factorize:

$$\mathsf{C} \Rightarrow \mathsf{A} \cong \mathsf{C} \Rightarrow \mathsf{B} \Rightarrow \mathsf{A}$$

and dually for the casts out of A. Remembering that all casts can be factorized through the dynamic type, it is sufficient to show that this is true for the embeddings and projections. Then we can say that A is *semantically more precise* than B, $\mathsf{A} \sqsubseteq^{\mathrm{sem}} \mathsf{B}$ when there is some ep pair, $e, p : \mathsf{A} \triangleleft \mathsf{B}$ satisfying

$$\mathsf{A}^{\mathsf{e}} \cong e; \mathsf{B}^{\mathsf{e}} \quad \text{and} \quad \mathsf{A}^{\mathsf{e}} \cong \mathsf{B}^{\mathsf{p}}; p$$

Note that this ep pair, if it exists, is unique because B^{e} is injective, respectively B^{p} is surjective. We prove that all derivations $\mathsf{d} :: \mathsf{A} \sqsubseteq \mathsf{B}$ produce ep pairs with this property and as a corollary we have coherence of the ep pair semantics of type precision (§5.3).

This property shows that the usual rules of type precision, far from being set in stone, are reflective of the way different types are embedded in the dynamic type. For instance, the famous function congruence rule, that lent type precision the name naïve subtyping:

$$\frac{\mathsf{A} \sqsubseteq \mathsf{A}' \qquad \mathsf{B} \sqsubseteq \mathsf{B}'}{\mathsf{A} \to \mathsf{B} \sqsubseteq \mathsf{A}' \to \mathsf{B}'}$$

is usually true because all functions casts factor through $? \to ?$, and the rule would be *false* if instead every function type $\mathsf{A} \to \mathsf{B}$ was given its own tag on the dynamic type, giving a less flexible but more easily implemented gradual language.

Semantic Term Precision: The Gradual Guarantee. What reasoning principles does a programmer gain from casts being constructed from ep pairs? We argue that they produce *precisely* the gradual guarantee. Recall that an ep pair from $e, p : \mathsf{A} \triangleleft \mathsf{B}$ are functions that satisfy retraction $p(e(x)) \cong x$ and projection $e(p(y)) \sqsubseteq_{\mathrm{err}} y$. But what do we mean by $\cong$ and $\sqsubseteq_{\mathrm{err}}$? The definition of equivalence $\cong$ is clear: we can use the well-established notion of contextual equivalence, that two programs are indistinguishable when run in the same context. For $\sqsubseteq_{\mathrm{err}}$, we can use a modified form of contextual *approximation*, that says that if $\mathsf{t} \sqsubseteq_{\mathrm{err}} \mathsf{t}'$ then under any context C, either $\mathsf{C}[\mathsf{t}]$ errors or $\mathsf{C}[\mathsf{t}], \mathsf{C}[\mathsf{t}']$ have equivalent behavior: they both terminate or they both diverge. This cleanly formalizes what it means for one term to have "more errors" than another, and we verify that all type precision judgments induce ep pairs with respect to this ordering.

The gradual guarantee can then be seen as a way to take the projection property $p(e(y)) \sqsubseteq_{\mathrm{err}} y$ and extend it to more general terms. To make this precise, we define a syntactic *term precision* judgment $t \sqsubseteq t'$, which extends type precision to terms. Syntactically, it can be defined by saying that t, t' have the same type erasure and that for all parallel type annotations $u@A$ in t and $u'@A'$ in t', we have $A \sqsubseteq A'$. This implies that if $\Gamma \vdash t : \mathsf{A}$ and $\Gamma' \vdash t' : \mathsf{A}'$, that $\Gamma \sqsubseteq \Gamma'$ pointwise and $\mathsf{A} \sqsubseteq \mathsf{A}'$. Then the gradual guarantee can now be stated succinctly by saying that if $t \sqsubseteq t'$, then $t \sqsubseteq_{\mathrm{err}} t'$.

The upside of the gradual guarantee is that it gives programmers a local-to-global reasoning principle about their programs: it takes a local change to the program, making certain typing annotations more or less precise, and gives a fact about the global behavior of the program in which the change was made: the only difference is the presence or absence of type errors. The global nature of the gradual guarantee makes it inherently difficult to maintain under language extension; we discuss the challenges in §8.

When it comes to proving the gradual guarantee for our language, our decomposition of casts into ep pairs again pays dividends. Since the definition of $\sqsubseteq_{\text{err}}$ quantifies over all syntactic contexts, a direct proof of the gradual guarantee by induction on terms goes nowhere. Instead, we use the standard weapon-of-choice for proving contextual approximation results: a logical relation. Just like our ep pair semantics, the logical relation is indexed by type precision derivations—specifically, a derivation $d :: A \sqsubseteq B$ is interpreted as a relation $[\![d]\!] \subset A \times B$. The intuition is that these relations are a "cross-type" analogue of $\sqsubseteq_{\text{err}}$. In order to prove that term precision is sound for this relation, we show that the logical relation is equivalent to a canonical relation defined by the ep pair. Since the ep pairs have been decomposed into categorical primitives (identity, composition, application of functors), the proof of the equivalence between the logical relation and the ep pair relation is a triviality. Our analysis shows that this proof technique by logical relation holds a canonical status and should scale well to more complex or exotic forms of gradual typing, providing a robust, reusable method for proving the gradual guarantee.

3 SURFACE LANGUAGE

We start by presenting a simple gradually typed surface language. Our focus is on the semantics and not gradual type checking, so we start with a fully annotated syntax that could serve as the output of a type inference/checking algorithm. This puts our language in a kind of middle ground between typical gradual surface languages and cast calculi. The syntax of types and terms is presented in Figure 1, and the typing derivations are presented in Figure 2. The general flavor of our language is captured by function introduction and elimination rules. Similar to [25], introduction forms produce precise type information, and elimination forms are mediated by a cast. Since the focus of this paper is on the semantics of casts and *not* gradual type checking, we allow many programs that would never be allowed by a gradual type checker, for instance, a program that includes a cast $\mathsf{A} \to \mathsf{B} \Rightarrow \mathbb{N}$ that always fails. While a smart type checker would rule this out, it has a perfectly well-defined semantics, and in practice such casts are macro-expressible in the sense of [9]. For example while $(\lambda x.x) + 1$ would not type check in the gradual lambda calculus, a local η-expansion is enough to fool the type checker: $((\lambda f : ?.f)(\lambda x.x)) + 1$, so we allow this cast syntactically, with syntax $\mathsf{add1}\,(\lambda \mathsf{x}.\, \mathsf{x} \mathbin{@} ? \to ?)$.

We include an isomorphism between $?$ and a sum of all the "type tags". We also include exlpicit syntax for the embeddings $\Uparrow^{?}_{\mathsf{A}}(\mathsf{t})$ and projections $\Downarrow^{?}_{\mathsf{A}}(\mathsf{t})$ since they have a natural semantics. To show the wider applicability of our semantics, we also include a simple form of objects, called *gradual records* based on [13]. A *precise* record type is of the form $\{\eta_1 : \mathsf{A}_1, \ldots, \eta_n : \mathsf{A}_n\}$ and acts as an ordinary record type: value forms simply supply all of the fields and elimination forms are projections at each of the defined fields $\eta_1, \ldots, \eta_n$. A *gradual* record type is of the form $\{\eta_1 : \mathsf{A}_1, \ldots, \eta_n : \mathsf{A}_n, ?\}$ (note the $?$!). A gradual record is a more dynamically typed form of record, in addition to supporting all of the fields $\eta_1 : \mathsf{A}_1, \ldots, \eta_n : \mathsf{A}_n$, a gradual record *might* support any other field η' not in that list. Whereas projection at the precise fields is a total operation, projection at the other fields may result in a (type) error. Following ([13]), we only present introduction and elimination forms for precise records, values of gradual record type are produced using casts from precise record type, and eliminated using the same projection form as precise records. This renders gradual records somewhat superfluous syntactically, but they play a large role in the semantics of the casts for syntactic records, explaining *why* casts like $\{\eta : \mathsf{A}, \eta' : \mathsf{A}'\} \Rightarrow \{\eta : \mathsf{A}\}$ do not error.

$$\begin{array}{llll} \text{Types} & A, B, C & ::= & ? \mid \mathbb{N} \mid 0 \mid A + B \mid 1 \mid A \times B \mid A \to B \mid \{\rho\} \mid \{\rho, ?\} \\ \text{Names} & \eta & \in & \mathfrak{F} \\ \text{Rows} & \rho & ::= & \eta : A, \ldots, \eta : A \end{array}$$

Fig. 1. Surface Language: Types

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \Uparrow^{?}_{A} (t) : ?} \qquad \frac{\Gamma \vdash t : ?}{\Gamma \vdash \Downarrow^{?}_{A} (t) : A} \qquad \frac{\Gamma \vdash t : 1 + \mathbb{N} + (? + ?) + (? \times ?) + \{?\} + (? \to ?)}{\Gamma \vdash \mathsf{roll}_{?}\, t : ?}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{unroll}_{?}\, (t \mathbin{@} A) : 1 + \mathbb{N} + (? + ?) + (? \times ?) + \{?\} + (? \to ?)} \qquad \frac{\Gamma, x : A \to B, y : A \vdash t : B}{\Gamma \vdash \mathsf{rec}\, x\, y \,.\, t : A \to B}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{case}\, (t \mathbin{@} A \Rightarrow 0)\, \mathsf{of}\, () : B} \qquad \frac{\forall i \in 1 \ldots, n.\ \Gamma \vdash t_i : A_i}{\Gamma \vdash \{\eta_1 \mapsto t_1, \ldots, \eta_n \mapsto t_n\} : \{\eta_1 : A_1, \ldots, \eta_n : A_n\}}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash (t \mathbin{@} A \Rightarrow \{\eta : B\}).\eta : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A_t \qquad \Gamma \vdash u : A_u}{\Gamma \vdash (t \mathbin{@} A_t \Rightarrow A_u \to B)\; u : B}$$

Fig. 2. Surface Typing Derivations (Fragment)

3.1 Type Precision

How can we show that our language is *gradual*? A key goal of any gradual language is to facilitate a smooth transition between typed and untyped modes of programming. For instance, if the type checker rejects a program, the user can change some types to be "more dynamic" and the type checker will allow it. It is a hallmark of [25]-style gradual typing (as opposed to the style of [31]) to support *fine-grained* interaction between typed and untyped components. That is, we have not only "fully typed" and "fully dynamic" but also a middle ground of "partially typed". For instance, we might know that an API produces lists, but don't yet want to commit to fixing a particular type for its elements. We could type this as $\mathsf{Listof}(?)$. Later on, we may decide that the list only contains functions $\mathsf{Listof}(? \to ?)$ and later that the functions will only be applied to numbers $\mathsf{Listof}(\mathbb{N} \to ?)$ and so on. This reasoning about certain types being "more dynamic" than others is captured by the *type precision* judgment $A \sqsubseteq B$, read A is more precise than B.

We present type precision in Figure 3 in a highly non-standard way. We view type precision derivations as having constructive content by defining certain ep pairs, so we give them a term syntax: $d :: A \sqsubseteq B$ can be read as d is a proof of $A \sqsubseteq B$, or as d is an ep pair from A to B. Taking this constructive view affects our choice of rules a great deal, since we want each rule to have a simple semantics. In particular, we adhere to the following compositionality principle:

Definition 3.1. Type Precision Compositionality Principle
The semantics of a type precision rule should be a function from the ep pairs produced by the premises to an ep pair for the conclusion.

This means that no rule can be justified by doing an induction on the *types* involved. For example, typically there is an explicit rule making $?$ the most imprecise type: $A \sqsubseteq ?$. However, constructively, the reason that $A \sqsubseteq ?$ holds can be wildly different for different types, so the only way to justify this rule would be to do induction on the type A. Instead of having this be a primitive rule of our type precision system, we make it a *derivable rule* for every A, and that derivation describes the casts involving A. To make this rule derivable, we introduce a cut/transitivity rule into the system, whose semantics is composition of ep pairs. There is an analogy with sequent calculus, which does not have cut and identity defined for all types, but instead these are admissible rules. The proof

$$
\begin{array}{llcl}
\text{Type Tags} & G & ::= & \mathbb{N} \mid ?+? \mid 1 \mid ?\times ? \mid ? \to ? \mid \{?\} \\
\text{Type Precision} & d & ::= & \mathrm{id}_A \mid d;d \mid 0_A \mid \mathrm{Tag}_G \mid d+d \mid d\times d \mid d \to d \\
 & & \mid & \{d\} \mid \mathrm{PrecGrad}\langle\rho\rangle \mid \{d,?\} \mid \mathrm{Width}_?\langle\rho;\eta,\ldots,\eta\rangle \\
\text{Row Precision} & d_\rho & ::= & \eta_1 : d_1, \ldots, \eta_n : d_n \\
\text{Environment Precision} & d_\Gamma & ::= & \cdot \mid d_\Gamma, d
\end{array}
$$

$$
\frac{}{\mathrm{id}_A :: A \sqsubseteq A}
\qquad
\frac{d :: A \sqsubseteq B \qquad d' :: B \sqsubseteq C}{d;d' :: A \sqsubseteq C}
\qquad
\frac{}{\mathrm{Tag}_G :: G \sqsubseteq ?}
$$

$$
\frac{d :: A \sqsubseteq A' \qquad d' :: B \sqsubseteq B'}{d + d' :: A + B \sqsubseteq A' + B'}
\qquad
\frac{d :: A \sqsubseteq A' \qquad d' :: B \sqsubseteq B'}{d \times d' :: A \times B \sqsubseteq A' \times B'}
\qquad
\frac{d :: A \sqsubseteq A' \qquad d' :: B \sqsubseteq B'}{d \to d' :: A \to B \sqsubseteq A' \to B'}
$$

$$
\frac{}{0_A :: 0 \sqsubseteq A}
\qquad
\frac{d_\rho :: \rho \sqsubseteq \rho'}{\{d_\rho\} :: \{\rho\} \sqsubseteq \{\rho'\}}
\qquad
\frac{}{\mathrm{PrecGrad}\langle\rho\rangle :: \{\rho\} \sqsubseteq \{\rho,?\}}
$$

$$
\frac{d_\rho :: \rho \sqsubseteq \rho'}{\{d_\rho,?\} :: \{\rho,?\} \sqsubseteq \{\rho',?\}}
\qquad
\frac{}{\mathrm{Width}_?\langle\rho;\eta_1,\ldots,\eta_n\rangle :: \{\rho,\eta_1 : ?,\ldots,\eta_n : ?,?\} \sqsubseteq \{\rho,?\}}
$$

Fig. 3. Type Precision

that cut and identity are admissible (cut elimination/identity expansion) then gives us valuable insight into the dynamics of the system. Here, by making the judgment $A \sqsubseteq ?$ derivable and not primitive, we reveal the underlying structure of the casts.

In fact there are four different "reasons" that $A \sqsubseteq ?$ may hold in our system. First, there is the identity $\mathrm{id}_?$ which represents the identity ep pair. Second, for each of the "tags" on the dynamic type, there is a cast Tag_G, that for the typical case where $?$ is implemented as a tagged union of the tag types, consists of an embedding that tags the value and a projection that checks for the right tag and otherwise errors. Third, there is the composition rule $d;d'$, where $d' :: B \sqsubseteq ?$, which is used with the tagging rule on the right for all of the connectives: products, sums, functions, records. Finally, there is the empty type rule $0_A :: 0 \sqsubseteq A$, whose embedding is the unique function out of the empty type and projection always errors. This makes 0 the least type by type precision, but unlike $?$, we make this fact a rule in the system because $0_A :: 0 \sqsubseteq A$ is implemented uniformly in A.

Next, we see a similar rule for many different types, the *functoriality* or *congruence rules* $d \to d'$, $d + d'$, $d \times d'$, $\{d\}$, $\{d,?\}$, which are often considered the *fundamental* rules of type precision, when combined with $A \sqsubseteq ?$. As we explain in §4.2, this makes sense semantically because all of these type connectives are *functors* (some covariant and some contravariant), and that co- and contravariant functors induce covariant actions on ep pairs.

We also decompose the rules for precise and gradual record precision, as described earlier in §2. For convenience, we define *row precision* $\rho \sqsubseteq \rho'$ as the field-wise lift of type precision. Our compositionality principle affects the design of the *width* precision rule for gradual rows. By analogy with width *subtyping*, a more obvious rule would be

$$\{\rho, \eta_1 : A_1, \ldots, \eta_n : A_n\} \sqsubseteq \{\rho\}$$

But as we see in §4.2, the embedding would need to cast A_i to $?$, and this would violate compositionality. Finally, we construct the canonical derivation of $A \sqsubseteq ?$ in Figure 4 by recursion on A.

$$
\begin{array}{rcl}
\mathsf{Imp}(?) & = & \mathsf{id}_? \\
\mathsf{Imp}(\mathbb{N}) & = & \mathsf{Tag}_{\mathbb{N}} \\
\mathsf{Imp}(1) & = & \mathsf{Tag}_1 \\
\mathsf{Imp}(0) & = & 0_?
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{Imp}(A + B) & = & (\mathsf{Imp}(A) + \mathsf{Imp}(B));\mathsf{Tag}_{?+?} \\
\mathsf{Imp}(A \times B) & = & (\mathsf{Imp}(A) \times \mathsf{Imp}(B));\mathsf{Tag}_{?\times?} \\
\mathsf{Imp}(A \to B) & = & (\mathsf{Imp}(A) \to \mathsf{Imp}(B));\mathsf{Tag}_{?\to?}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Imp}(\{\eta_1 : A_1, \ldots, \eta_n : A_n\}) & = & \{\eta_1 : \mathsf{Imp}(A_1), \ldots, \eta_n : \mathsf{Imp}(A_n)\};\mathsf{PrecGrad}\langle \eta_1 : ?, \ldots, \eta_n : ?\rangle; \\
 & & \mathsf{Width}_?\langle \cdot ; \eta_1, \ldots, \eta_n \rangle;\mathsf{Tag}_{\{?\}} \\
\mathsf{Imp}(\{\eta_1 : A_1, \ldots, \eta_n : A_n, ?\}) & = & \{\eta_1 : \mathsf{Imp}(A_1), \ldots, \eta_n : \mathsf{Imp}(A_n), ?\};\mathsf{Width}_?\langle \cdot ; \eta_1, \ldots, \eta_n \rangle;\mathsf{Tag}_{\{?\}}
\end{array}
$$

Fig. 4. Dynamic Type Imprecision

$$
\begin{array}{llcl}
\text{Types} & \mathbf{A}, \mathbf{B} & ::= & ? \mid \mathbb{N} \mid \mathbf{0} \mid \mathbf{A} + \mathbf{B} \mid \mathbf{A} \to \mathbf{B} \mid \times_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_i (\mathbf{I} \text{ finite}) \mid \prod_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_i \\
\text{Values} & \mathbf{v} & ::= & \mathbf{roll}_? \, \mathbf{v} \mid \mathbf{rec}\, \mathbf{x}\, \mathbf{y} \,.\, \mathbf{t} \mid \lambda \mathbf{x}.\, \mathbf{t} \mid \langle \mathbf{i}_1 = \mathbf{v}_1, \ldots, \mathbf{i}_n = \mathbf{v}_n \rangle_{\mathbf{I}} \mid \{\mathbf{i}_1 = \mathbf{v}_1, \ldots, \mathbf{i}_n = \mathbf{v}_n\}_{\mathbf{I}} \cdots \\
\text{Partial Values} & \mathbf{v}_{\mho} & ::= & \mathbf{v} \mid \mho \\
\text{Evaluation Contexts} & \mathbf{E} & ::= & [\cdot] \mid \mathbf{roll}_? \, \mathbf{E} \mid \mathbf{unroll}_? \, \mathbf{E} \mid \mathbf{E}\, \mathbf{u} \mid \mathbf{v}\, \mathbf{E} \mid \langle \mathbf{i} = \mathbf{v}, \ldots, \mathbf{i} = \mathbf{t}, \ldots, \mathbf{i} = \mathbf{t} \rangle_{\mathbf{I}} \mid \\
 & & \mid & \{\mathbf{i} = \mathbf{v}, \ldots, \mathbf{i} = \mathbf{t}, \ldots, \mathbf{i} = \mathbf{t}\}_{\mathbf{I}} \mid \mathbf{E}|_{\mathbf{I} \subseteq \mathbf{J}} \mid \{\mathbf{t} ; \mathbf{i} = \mathbf{v}_{\mho}, \ldots\} \mid \cdots \\
\text{Contexts} & \mathbf{C} & ::= & [\cdot] \mid \mathbf{roll}_? \, \mathbf{C} \mid \mathbf{unroll}_? \, \mathbf{C} \mid \lambda \mathbf{x}.\, \mathbf{C} \mid \mathbf{C}\, \mathbf{u} \mid \mathbf{t}\, \mathbf{C} \mid \cdots
\end{array}
$$

Fig. 5. Core language terms, types (Fragment)

$$
\frac{}{\Gamma \vdash \mho : \mathbf{A}}
\qquad
\frac{\Gamma \vdash \mathbf{t} : \mathbf{1} + \mathbb{N} + (? + ?) + (? \times ?) + (? \to ?) + \prod_{- \in \mathfrak{F}} ?}{\Gamma \vdash \mathbf{roll}_? \, \mathbf{t} : ?}
$$

$$
\frac{\Gamma \vdash \mathbf{t} : ?}{\Gamma \vdash \mathbf{unroll}_? \, \mathbf{t} : \mathbf{1} + \mathbb{N} + (? + ?) + (? \times ?) + (? \to ?) + \prod_{- \in \mathfrak{F}} ?}
\qquad
\frac{\Gamma, \mathbf{x} : \mathbf{A}_1 \vdash \mathbf{t} : \mathbf{A}_2}{\Gamma \vdash \lambda \mathbf{x}.\, \mathbf{t} : \mathbf{A}_1 \to \mathbf{A}_2}
$$

$$
\frac{\Gamma \vdash \mathbf{t}_1 : \mathbf{A}_2 \to \mathbf{A} \qquad \Gamma \vdash \mathbf{t}_2 : \mathbf{A}_2}{\Gamma \vdash \mathbf{t}_1\, \mathbf{t}_2 : \mathbf{A}}
$$

$$
\frac{\forall 1 \le l < m \le n, \mathbf{i}_l \ne \mathbf{i}_m \qquad \{\mathbf{i}_1, \ldots, \mathbf{i}_n\} = \mathbf{I} \qquad \forall 1 \le l \le n, \Gamma \vdash \mathbf{t}_l : \mathbf{A}_{\mathbf{i}_l}}{\Gamma \vdash \langle \mathbf{i}_1 = \mathbf{t}_1, \ldots, \mathbf{i}_n = \mathbf{t}_n \rangle_{\mathbf{I}} : \times_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_i}
\qquad
\frac{\Gamma \vdash \mathbf{t} : \times_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_{\mathbf{i}} \qquad \mathbf{j} \in \mathbf{I}}{\Gamma \vdash \pi_{\mathbf{j}}\, \mathbf{t} : \mathbf{A}_{\mathbf{j}}}
$$

$$
\frac{\forall 1 \le l < m \le n, \mathbf{i}_l \ne \mathbf{i}_m \qquad \{\mathbf{i}_1, \ldots, \mathbf{i}_n\} \subseteq \mathbf{I} \qquad \forall 1 \le l \le n, \Gamma \vdash \mathbf{t}_l : \mathbf{A}_{\mathbf{i}_l}}{\Gamma \vdash \{\mathbf{i}_1 = \mathbf{t}_1, \ldots, \mathbf{i}_n = \mathbf{t}_n\}_{\mathbf{I}} : \prod_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_i}
\qquad
\frac{\Gamma \vdash \mathbf{t} : \prod_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_{\mathbf{i}} \qquad \mathbf{j} \in \mathbf{I}}{\Gamma \vdash \mathbf{t}.\mathbf{j} : \mathbf{A}_{\mathbf{j}}}
$$

$$
\frac{\Gamma \vdash \mathbf{t} : \prod_{\mathbf{i} \in \mathbf{J}} \mathbf{A}_i \qquad \mathbf{I} \subseteq \mathbf{J}}{\Gamma \vdash \mathbf{t}|_{\mathbf{I} \subseteq \mathbf{J}} : \prod_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_i}
\qquad
\frac{\Gamma \vdash \mathbf{t} : \prod_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_i \qquad \mathbf{J} = \{\mathbf{i}_1, \ldots, \mathbf{i}_n\} \qquad \mathbf{I} \# \mathbf{J} \qquad \forall \mathbf{j} \in \mathbf{J}.\, \Gamma \vdash \mathbf{v}_{\mho \mathbf{j}} : \mathbf{A}_{\mathbf{j}}}{\Gamma \vdash \{\mathbf{t} ; \mathbf{j}_1 = \mathbf{v}_{\mho 1}, \ldots, \mathbf{j}_n = \mathbf{v}_{\mho n}\} : \prod_{\mathbf{i} \in \mathbf{I} \uplus \mathbf{J}} \mathbf{A}_i}
$$

Fig. 6. Core Language Typing (Fragment)

4 CORE LANGUAGE AND ELABORATION

Rather than give the surface language an operational or denotational semantics *directly*, we elaborate into a non-gradual intrinsically typed core language. The advantage of such an approach is that the core language used has a standard operational semantics (and mostly standard denotational semantics), and so syntactic properties of the language are cleanly separated from the semantics of casts. If we were to construct an operational semantics for the surface language, we could use the semantics of the core language as a guide, ensuring that the two semantics are in some simulation relation.

The syntax of our core language terms and types is presented in Figure 5 and typing derivations in Figure 6. The language can be succinctly described as a call-by-value simply typed lambda calculus with general recursion, a single uncatchable error $\mho$, a recursive type $?$, and a partial product type $\prod_{i \in I} A_i$. We write the error as $\mho$ which can be raised at any type and is used for all type errors, we do not consider blame in this semantics. It is important for the gradual guarantee that the error is uncatchable, see §8. To interpret the dynamic type, we add a single recursive type to the language $?$. While in general the dynamic type does not have to be a tagged union of all the connectives, for concreteness we use that implementation here. The dynamic type has an isomorphism $\mathbf{roll}_? \cdot, \mathbf{unroll}_? \cdot$ exhibiting the dynamic type as a recursively defined sum. We design the dynamic type with our surface language in mind: there is one case for each of the type tags of the surface language.

To simplify the semantics of record types, we include a finite product type $\bigtimes_{i \in I} A_i$ where I is a finite indexing set and A_i is a type for each $i \in I$. We abbreviate $1 = \bigtimes_{i \in \emptyset} \cdot$ and $A_1 \times A_2 = \bigtimes_{i \in \{0,1\}} A_i$. The values of this type are written $\langle v_{i_1}, \ldots, v_{i_n} \rangle_I$ and consist of a value for each A_i and the side-conditions on the introduction rule ensure that every i is instantiated exactly once. The elimination forms are the apparent projections.

To encode the *partiality* of gradual record types, we include the partial product type $\prod_{i \in I} A_i$ where I is any set (in practice, countable), and A_i a type for each $i \in I$. The values $\{v_{i_1}, \ldots, v_{i_n}\}_I$ are similar to the values in a product type, except some fields can be left *undefined*. We write the projections in "method"-style $t.i$. To keep the syntax finite, a value of type $\prod_{i \in I} A_i$ can only have finitely many fields defined, in partial function terminology, we would say that the values are partial functions with finite support. We include two more operations on partial products: extension and weakening. Weakening $t|_{I \subseteq J}$ is the restriction of a partial function to a *subset* of its domain. Extension $\{t\,; j_1 = v_{\mho 1}, \ldots, j_n = v_{\mho n}\}$ allows for extending a partial product to a *superset* of its domain, a single extension $j = v_{\mho}$ either adds a value to the tuple or declares the field undefined by using $\mho$. We put a partial-value restriction on the extensions to keep the operational semantics simple.

Like the dynamic type, we could express $\prod_{i \in I} A_i$ using other connectives if we had a more expressive type system. The type $\bigtimes_{i \in I} 1 \to A_i$ is close, but includes "fields" that when projected diverge, and if we added other effects such as printing, then these would be able to print as well. If we had a more fine-grained effect system, we could encode this, but we avoid this to keep the core language simple.

4.1 Operational Semantics and Contextual Approximation

We define a call-by-value operational semantics in Figure 7, using evaluation contexts in the style of [10]. The most rules are standard and we elide them. The error propagates across all evaluation contexts since it is uncatchable. The projection for the partial product errors if the field is not present. Weakening simply removes the forgotten fields. Extension is the only rule that is not an axiom, essentially performing induction on the fields introduced. If the extension is $\mho$, it acts as a *strengthening*. If the extension is a value, it is added to the tuple. Finally when the extensions are exhausted, the tuple is returned.

To prove that our ep pairs satisfy the projection property $e(p(x)) \sqsubseteq_{\text{err}} x$, we first need to clarify what $\sqsubseteq_{\text{err}}$ means. Intuitively $t \sqsubseteq_{\text{err}} u$ should mean that t has the same behavior as u except that it "errors more". To make this precise, we start with a simple case, when the programs have unit type, it means that if t terminates (resp. diverges), then u terminates (diverges), and if t errors, u might do anything:

$$\frac{}{E[\mho] \longmapsto \mho} \qquad \frac{}{E[\mathbf{unroll}_? \, \mathbf{roll}_? \, v] \longmapsto E[v]} \qquad \frac{}{E[\pi_{i_k} \langle i_1 = v_{i_1}, \ldots, i_n = v_{i_n} \rangle_I] \longmapsto E[v_{i_k}]}$$

$$\frac{i \in \{i_1, \ldots, i_n\}}{E[\{i_1 = v_{i_1}, \ldots, i_n = v_{i_n}\}_I.i] \longmapsto E[v_i]} \qquad \frac{i \notin \{i_1, \ldots, i_n\}}{E[\{i_1 = v_1, \ldots, i_n = v_n\}_I.i] \longmapsto E[\mho]}$$

$$\frac{J = \{j_1, \ldots, j_n\}}{E[\{i_1 = v_{j_1}, \ldots, i_n = v_{j_n}\}_I|_{J \subseteq I}] \longmapsto E[\{j_1 = v_{j_1}, \ldots, j_n = v_{j_n}\}_J]}$$

$$\frac{E[\{\{\ldots\}_{I \cup \{j_1\}} ; j_2 = v_{\mho j_2}, \ldots\}] \longmapsto E[v']}{E[\{\{i_1 = v_1, \ldots, i_n = v_n\}_I ; j_1 = \mho, j_2 = v_{\mho j_2}, \ldots\}] \longmapsto E[v']}$$

$$\frac{E[\{\{\ldots, j_1 = v_{j_1}\}_{I \cup \{j_1\}} ; j_2 = v_{\mho j_2}, \ldots\}] \longmapsto E[v']}{E[\{\{\ldots\}_I ; j_1 = v_{j_1}, j_2 = v_{\mho j_2}, \ldots\}] \longmapsto E[v']} \qquad \frac{}{E[\{\{\ldots\}_I ; \cdot\}] \longmapsto E[\{\ldots\}_I]}$$

Fig. 7. Core Language Operational Semantics (Fragment)

Definition 4.1. Error Approximation We say a closed program $\cdot \vdash t : 1$ approximates $\cdot \vdash u : 1$, written $t \sqsubseteq_{\text{err}} u$ when the following hold:

(1) If $t \longmapsto^* \langle\rangle$, then $u \longmapsto^* \langle\rangle$.
(2) If $C[t] \Uparrow$, then $C[u] \Uparrow$.

Note that this is different from the usual notion of approximation, which captures when one term *diverges more*.

Then we extend this notion to open terms of arbitrary type using *contexts*. That is, we say t contextually approximates u if whenever they are placed into the same program, the program with t approximates the program with u:

Definition 4.2. Core Language Contextual Error Approximation We say $\Gamma \vdash t : A$ (contextually) approximates $\Gamma \vdash u : A$, written $\Gamma \vdash t \sqsubseteq^{\text{ctx}}_{\text{err}} u : A$ or simply $t \sqsubseteq^{\text{ctx}}_{\text{err}} u$ if for any closing context $C : (\Gamma \vdash A) \Rightarrow (\cdot \vdash 1)$, $C[t] \sqsubseteq_{\text{err}} C[u]$.

We then define *contextual equivalence* in the usual way as approximation both ways. Though our notion of approximation is non-standard, the induced equivalence coincides with the usual definition:

Definition 4.3. Core Language Contextual Equivalence We say t is contextually equivalent to u when $t \sqsubseteq^{\text{ctx}}_{\text{err}} u$ and $u \sqsubseteq^{\text{ctx}}_{\text{err}} t$.

As usual, the restriction to the context returning a value of 1 is artificial, in fact this definition easily implies a similar version where 1 is any "ground" (i.e., not higher order) type by writing an extended context.

Finally, in order to explain our theorems to a surface language programmer without expecting them to understand the core language, we should formulate notions of contextual error approximation and equivalence quantifying over *surface* contexts. However, since any operational semantics for the surface language should have to be in simulation with the operational semantics for the core language, it is clear that the notions of contextual approximation and equivalence in the core language are *stronger* than those in the surface language. This justifies our abuse of notation to define approximation and equivalence of source terms as just approximation (resp. equivalence) of their elaborations.

With approximation and equivalence now defined precisely, we can define ep pairs as described in §1. For syntactic convenience, we will define embedding-projection pairs as evalation contexts, rather than closed terms of function type

Definition 4.4. Embedding-Projection Pair We say $\mathbf{E}_e[\cdot : \mathbf{A}] : \mathbf{B}, \mathbf{E}_p[\cdot : \mathbf{B}] : \mathbf{A}$ with form an embedding-projection pair from $\mathbf{A}$ to $\mathbf{B}$, written $\mathbf{E}_e, \mathbf{E}_p :: \mathbf{A} \triangleleft \mathbf{B}$ when $\mathbf{E}_p[\mathbf{E}_e\cdot] \approx^{\text{ctx}} [\cdot]$ and $\mathbf{E}_e[\mathbf{E}_p\cdot] \sqsubseteq^{\text{ctx}}_{\text{err}} [\cdot]$.

4.2 Elaboration

Next, we define the elaboration of the surface into the core language. First the elaboration for types, type precision and casts is defined in Figure 8. Every surface type A denotes a type A^+ in the core language. Since the languages are so similar, the type translation is completely obvious except for the record types. A precise record is implemented as a tagged product. A gradual record is implemented as a pair of a tagged product of its precise fields with a partial product of the rest of the remaining fields at the dynamic type.

Next, we interpret the type precision derivations $d :: A \sqsubseteq B$ as ep pairs $d^{\mathbf{e}}, d^{\mathbf{p}} :: A^+ \triangleleft B^+$. Since the constructions on embeddings and projections are the same for many rules, we use the shorthand $d^{\mathbf{e},\mathbf{p}}$ to define them both at the same time. Note that in the function rule, the projection and embedding are swapped, denoted $d^{\mathbf{p},\mathbf{e}}$. Since we've abided by the compositionality principle (Definition 3.1), the semantics of each rule are very simple The identity rule is the identity ep pair, and the composition is the composition of ep pairs, with embeddings composed one way and projection the opposite. The tagging embedding applies the tag and the tagging projection errors if the wrong tag is encountered. The functorial rules use the actions defined in Figure 9. A syntactic intuition is that they are all essentially η-expansions with the functions inserted. The product action uses an arbitrary order on the field names, but since ep pairs can only ever error (see §5) and there is only one error any order would produce an equivalent term. If blame were present, we could introduce an arbitrary choice or simply collect all errors.

With elaboration of type precision derivations defined, we can elaborate casts, first by defining the embedding and projection for a type using the canonical derivation $\text{Imp}(A) :: A \sqsubseteq ?$ and then defining an arbitrary cast to be an upcast followed by a downcast.

Then the elaboration for terms (Figure 10) is completely systematic: introduction forms are translated to introduction forms, elimination forms are translated with casts inserted.

4.3 Soundness Theorems

While we do not yet have the means to *prove* contextual approximation of terms in our core language (we will rectify this in §5), we will state now the soundness theorems for our elaboration and give some intuition as to why they are true.

Types are EP Pairs. First, we should verify that our so-called embedding-projection pairs deserve the name:

Theorem 4.5. *Type Precision Derivations denote Embedding-Projection Pairs*
If $d :: A \sqsubseteq B$*, then* $d^{\mathbf{e}}, d^{\mathbf{p}} : A^+ \triangleleft B^+$.

Which means that our interpretation of surface types as embedding-projection pairs is sound.

Corollary 4.6. *Surface Types are Core Language Embedding-Projection Pairs*
For every A, $A^{\mathbf{e}}, A^{\mathbf{p}} : A^+ \triangleleft B^+$.

Semantic Type Precision. Next, we consider semantic type precision. We have given an elaboration semantics for *derivations* of $A \sqsubseteq B$, but existing intuition is that $A \sqsubseteq B$ is a mere proposition, so we should be able to define a *meaning* for $A \sqsubseteq B$ for which any derivation $d :: A \sqsubseteq B$ serves as

$$
\begin{aligned}
?^{+} &\stackrel{\text{def}}{=} ?\\
\mathbb{N}^{+} &\stackrel{\text{def}}{=} \mathbb{N}\\
(A_1 \to A_2)^{+} &\stackrel{\text{def}}{=} {A_1}^{+} \to {A_2}^{+}\\
(\{\eta_1 : A_{\eta_1}, \ldots, \eta_n : A_{\eta_n}\})^{+} &\stackrel{\text{def}}{=} \bigtimes_{\eta \in \{\eta_1,\ldots,\eta_n\}} {A_\eta}^{+}\\
(\{\eta_1 : A_{\eta_1}, \ldots, \eta_n : A_{\eta_n}, ?\})^{+} &\stackrel{\text{def}}{=} \bigtimes_{\eta \in \{\eta_1,\ldots,\eta_n\}} {A_\eta}^{+} \times \prod_{- \in \mathfrak{F} \setminus \{\eta_1,\ldots,\eta_n\}} ?\\
A^{\mathbf{e},\mathbf{p}} &\stackrel{\text{def}}{=} \mathrm{Imp}(A)^{\mathbf{e},\mathbf{p}}\\
(A \implies B)^{+} &\stackrel{\text{def}}{=} B^{\mathbf{p}}[A^{\mathbf{e}}]
\end{aligned}
$$

$$
\begin{aligned}
{\mathrm{id}_A}^{\mathbf{e},\mathbf{p}} &\stackrel{\text{def}}{=} [\cdot]\\
(d;d')^{\mathbf{e}} &\stackrel{\text{def}}{=} d'^{\mathbf{e}}[d^{\mathbf{e}}]\\
(d;d')^{\mathbf{p}} &\stackrel{\text{def}}{=} d^{\mathbf{p}}[d'^{\mathbf{p}}]\\
{\mathrm{Tag}_G}^{\mathbf{e}} &\stackrel{\text{def}}{=} \mathbf{inj}_{G^+}\ [\cdot]\\
{\mathrm{Tag}_G}^{\mathbf{p}} &\stackrel{\text{def}}{=} \mathbf{case}\,[\cdot]\,\mathbf{of}\\
&\qquad \mathbf{inj}_{G^+}\ x.\, x\\
&\qquad \mid \mathbf{else}.\ \mho\\
(d + d')^{\mathbf{e},\mathbf{p}} &\stackrel{\text{def}}{=} d^{\mathbf{e},\mathbf{p}} + d'^{\mathbf{e},\mathbf{p}}\\
(d \times d')^{\mathbf{e},\mathbf{p}} &\stackrel{\text{def}}{=} d^{\mathbf{e},\mathbf{p}} \times d'^{\mathbf{e},\mathbf{p}}\\
(d \to d')^{\mathbf{e},\mathbf{p}} &\stackrel{\text{def}}{=} d^{\mathbf{p},\mathbf{e}} \to d'^{\mathbf{e},\mathbf{p}}
\end{aligned}
$$

$$
\begin{aligned}
\{\eta_1 : d_1, \ldots, \eta_n : d_n\}^{\mathbf{e},\mathbf{p}} &\stackrel{\text{def}}{=} \bigtimes_{i \in 1,\ldots,n} {d_i}^{\mathbf{e},\mathbf{p}}\\
\{\eta_1 : d_1, \ldots, \eta_n : d_n, ?\}^{\mathbf{e},\mathbf{p}} &\stackrel{\text{def}}{=} (\{\eta_1 : d_1, \ldots, \eta_n : d_n\}^{\mathbf{e},\mathbf{p}}) \times [\cdot]\\
\mathrm{PrecGrad}\langle\rho\rangle^{\mathbf{e}} &\stackrel{\text{def}}{=} \langle [\cdot], \{\}_{\mathfrak{F}-\rho}\rangle\\
\mathrm{PrecGrad}\langle\rho\rangle^{\mathbf{p}} &\stackrel{\text{def}}{=} \pi_1\,[\cdot]\\
\mathrm{Width}_?\langle\rho;\eta_1,\ldots,\eta_n\rangle^{\mathbf{e}} &\stackrel{\text{def}}{=} \mathbf{let}\,\langle x_p, x_g\rangle = [\cdot]\,\mathbf{in}\\
&\qquad \mathbf{let}\,x_{\eta_1} = \pi_{\eta_1}\,x_p \cdots\ \mathbf{in}\\
&\qquad \left\langle \begin{array}{l} \langle \eta_1' = \pi_{\eta_1'}\,x_p, \ldots\rangle_{\rho},\\ \quad \{x_g\,;\eta_1 = x_{\eta_1}, \ldots\} \end{array} \right\rangle\\
\mathrm{Width}_?\langle\rho;\eta_1,\ldots,\eta_n\rangle^{\mathbf{p}} &\stackrel{\text{def}}{=} \mathbf{let}\,\langle x_p, x_g\rangle = [\cdot]\,\mathbf{in}\\
&\qquad \left\langle \begin{array}{l} \left\langle \begin{array}{l} \eta_1' = \pi_{\eta_1'}\,x_p, \ldots,\\ \eta_1 = x_g.\eta_1, \ldots \end{array} \right\rangle_{\rho,\eta_1,\ldots},\\ \quad x_g|_{\mathfrak{F}-\rho,\eta_1,\ldots \subseteq \mathfrak{F}-\rho} \end{array} \right\rangle
\end{aligned}
$$

Fig. 8. Source to Core Type and Type Precision Elaborator (Fragment)

$$
\begin{aligned}
(E_1[\cdot : A_1] : B_1) + (E_2[\cdot : A_2] : B_2) &= \mathbf{case}\,([\cdot] : A_1 + A_2)\,\mathbf{of}\,x_1.\,\mathbf{inj_1}\,E_1[x_1] \mid x_2.\,\mathbf{inj_2}\,E_2[x_2] : B_1 + B_2\\
(E_1[\cdot : A_1] : B_1) \to (E_2[\cdot : A_2] : B_2) &= \mathbf{let}\,x_f = ([\cdot] : B_1 \to A_2)\,\mathbf{in}\,\lambda x_1.\,E_2(x_f\,(E_1[x_1])) : A_1 \to B_2\\
\bigtimes_{i \in 1,\ldots,n} E_i[\cdot : A_i] : B_i &= \mathbf{let}\,y = [\cdot]\,\mathbf{in}\,\mathbf{let}\,x_1 = \pi_1\,x_1\,\mathbf{in} \cdots \langle i_1 = x_1, \ldots, i_n = x_n\rangle_{\{1,\ldots,n\}}
\end{aligned}
$$

Fig. 9. Functorial Action of Type Connectives

$$
\begin{aligned}
(x)^{+} &\stackrel{\text{def}}{=} \mathbf{x}\\
(\Uparrow^{?}_{A}\ (t))^{+} &\stackrel{\text{def}}{=} A^{\mathbf{e}}[(t)^{+}]\\
(\Downarrow^{?}_{A}\ (t))^{+} &\stackrel{\text{def}}{=} A^{\mathbf{p}}[(t)^{+}]
\end{aligned}
\qquad
\begin{aligned}
(\mathrm{roll}_?\ t)^{+} &\stackrel{\text{def}}{=} \mathbf{roll}_?\ (t)^{+}\\
(\mathrm{unroll}_?\ (t\ @\ A))^{+} &\stackrel{\text{def}}{=} \mathbf{unroll}_?\ A^{\mathbf{e}}[(t)^{+}]\\
(\lambda x.\, t)^{+} &\stackrel{\text{def}}{=} \lambda \mathbf{x}.\ (t)^{+}\\
((t\ @\ A_t \Rightarrow A_u \to B)\ u)^{+} &\stackrel{\text{def}}{=} ((A_t \implies (A_u \to B))^{+}[(t)^{+}])\ u
\end{aligned}
$$

Fig. 10. Source to Core Term Elaborator (Fragment)

evidence. Clearly, this meaning cannot just be "there exists an ep pair $A^{+} \triangleleft B^{+}$", because this would justify rules like $\mathbb{B} \sqsubseteq \mathbb{N}$ that we don't necessarily want. Instead, what works is saying "there exists an ep pair $A^{+} \triangleleft B^{+}$ satisfying a uniquely defining property", which has the side-effect of proving *coherence* of the semantics of type precision derivations, that any two derivations $d, d' :: A \sqsubseteq B$ of the same type precision judgment have the same semantics: $[\![d^{\mathbf{e}}]\!] \approx^{\mathrm{ctx}} [\![d'^{\mathbf{e}}]\!]$ and $[\![d^{\mathbf{p}}]\!] \approx^{\mathrm{ctx}} [\![d'^{\mathbf{p}}]\!]$.

As described in §2, the property that we want is that the ep pairs *factorize* the ep pairs for types. We can also formulate it purely as a *property* of the casts involved, which can be interpreted in any cast calculus.

Definition 4.7. Semantic Type Precision We say A is *semantically* more precise than B, written $A \sqsubseteq^{\mathrm{sem}} B$ when either of the following equivalent properties hold:

(1) There exists an ep pair $\mathbf{e}, \mathbf{p} : A \triangleleft B$ satisfying $A^{\mathbf{e}} \approx^{\mathrm{ctx}} \mathbf{e}; B^{\mathbf{e}}$, and $A^{\mathbf{p}} \approx^{\mathrm{ctx}} B^{\mathbf{p}}; \mathbf{p}$.

(2) The casts between A, B form an ep pair: $(A \Rightarrow B), (B \Rightarrow A) : A \triangleleft B$

Proof. To show the first implies the second, we prove the projection property, the retraction property is similar.

$$(B \Rightarrow A)^{+}; (A \Rightarrow A)^{+} \approx^{\mathrm{ctx}} B^{e}; A^{p}; A^{e}; B^{p} \approx^{\mathrm{ctx}} B^{e}; B^{p}; \mathbf{p}; \mathbf{e}; B^{e}; B^{p} \approx^{\mathrm{ctx}} \mathbf{p}; \mathbf{e} \sqsubseteq^{\mathrm{ctx}}_{\mathrm{err}} \mathrm{id}_{B}$$

To show the second implies the first, since embeddings have a unique projection and vice-versa, then by Lemma 4.6 it is sufficient to show that $(A \Rightarrow B; B^{e}), A^{p} : A^{+} \triangleleft ?^{+}$ and $A^{e}, (B^{p}; B \Rightarrow A) : A^{+} \triangleleft ?^{+}$. We prove the former, the latter is analogous. For the retraction property:

$$A \Rightarrow B; B^{e}; A^{p} \approx^{\mathrm{ctx}} A \Rightarrow B; B \Rightarrow A \approx^{\mathrm{ctx}} \mathrm{id}_{A}$$

by assumption, and for projection:

$$A^{p}; (A \Rightarrow B); B^{e} \approx^{\mathrm{ctx}} A^{p}; A^{e}; B^{p}; B^{e} \sqsubseteq^{\mathrm{ctx}}_{\mathrm{err}} \mathrm{id}_{?}$$

□

An intuitive result of this semantics is that if $A \sqsubseteq^{\mathrm{sem}} B$, then any cast involving A factors through B:

Theorem 4.8. *Semantic Precision implies Cast Factorization*
If $A \sqsubseteq^{sem} B$, *then* $A \Rightarrow C \approx^{ctx} A \Rightarrow B \Rightarrow C$ *and* $C \Rightarrow A \approx^{ctx} C \Rightarrow B \Rightarrow A$

This semantic definition justifies the following rules in some generality:

$$0 \sqsubseteq^{\mathrm{sem}} A \qquad\qquad A \sqsubseteq^{\mathrm{sem}} ?$$

the $?$ rule is essentially what we argued for in §1, and is included as a syntactic rule in every gradually typed language. The $0 \sqsubseteq^{\mathrm{sem}} A$ is more surprising/controversial, but follows naturally from the definition. First, the retraction property $(0 \Rightarrow A); (A \Rightarrow 0) \approx^{\mathrm{ctx}} \mathrm{id}_{0}$ should be true because all functions out of 0 are equivalent because they are dead code. Second, the projection property $(A \Rightarrow 0); (0 \Rightarrow A) \sqsubseteq^{\mathrm{ctx}}_{\mathrm{err}} \mathrm{id}_{A}$ should be true because $A \Rightarrow 0$ should always error on any A input. We consider how to rectify this rule with existing interpretations of gradual typing in §8.

Furthermore, for our specific interpretation of casts, we can prove all of the syntactic rules of type precision in §3 sound with this semantics, more proof details are in §5:

Theorem 4.9. *Type Precision implies EP Pair Factorization*
Given any type precision derivation $d :: A \sqsubseteq B$, $A^{e} \approx^{ctx} B^{e}[d^{e}]$ *and* $A^{p} \approx^{ctx} d^{p}[B^{e}]$, *and therefore* $A \sqsubseteq^{sem} B$

Finally, as a corollary we have *coherence* for the type precision semantics, because any ep pairs witnessing semantic type precision must be equal.

Corollary 4.10. *Type Precision Semantics is Coherent*
For any derivations $d, d' :: A \sqsubseteq B$, $d^{e} \approx^{ctx} d'^{e}$ *and* $d^{p} \approx^{ctx} d'^{p}$.

5 DENOTATIONAL SEMANTICS

In order to prove the soundness results from the previous section, we need a more tractable way to prove contextual approximation and equivalence of programs. This is usually achieved by either a logical relation based on the operational semantics, or constructing an adequate denotational semantics. In this paper we choose to develop a domain-theoretic denotational semantics to highlight the similarities (and differences) to traditional domain theoretic models of untyped lambda calculus. The relationship is not entirely trivial: in classical models dynamic type errors are interpreted as $\bot$, which is also the denotation of a diverging computation, whereas in gradual typing type errors should be distinct from divergence, for instance if we want a semantic proof of the gradual guarantee. Despite this difference, the construction of ep pairs is essentially the same as in classical domain theory, except using an error order rather than the "divergence" order.

To prove the soundness results we proceed in two steps. First, we develop a semantic analogue of the soundness theorem in question. For example, our domains come with an error ordering $\sqsubseteq_{\text{err}}$ that is the semantic analogue of contextual error approximation, so to prove that our embedding-projection pairs satisfy the retraction and projection properties, we first show that their denotations satisfy these properties with respect to $\sqsubseteq_{\text{err}}$. This is much easier, because we have compositional definitions of $\sqsubseteq_{[\![A]\!]}$ for each type, for instance for pairs $(x, y) \sqsubseteq_{X\times Y} (x', y')$ just when $x \sqsubseteq_X x'$ and $y \sqsubseteq_Y y'$. Then to get the corresponding *syntactic* result, we use the *adequacy* theorem for the denotational semantics. A denotational semantics is called *adequate* when approximation of denotations is sound for approximation of closed programs, that is if $[\![\mathbf{t}]\!] \sqsubseteq_{err}^{sem} [\![\mathbf{u}]\!]$ then $\mathbf{t} \sqsubseteq_{\text{err}}^{\text{ctx}} \mathbf{u}$. This theorem takes some work, but the techniques employed are well established [20, 28], so we relegate most details to the appendix.

5.1 Error Domains

In classical domain theory, a domain is equipped with a "definedness" ordering, where a least element $\perp$ represents a completely undefined computation. The undefined element serves as the interpretation of diverging computations. The undefined and partially defined elements are crucial in the construction of recursively defined domains, and the standard technique ([22, 28]) uses embedding-projection pairs in an essential way to construct non-well-founded domains.

If X, Y are domains, then an embedding-projection pair is a pair of continuous functions $e : X \longrightarrow Y$ and $p : Y \longrightarrow X$ with $p \circ e = \mathrm{id}_X$ and $e \circ p \leq \mathrm{id}_Y$. This means that for any $y \in Y$, $e(p(y))$ is a less well defined element that y. However, if we were to take this as our interpretation of casts for gradual types, this would mean that a failed cast *diverges* when the input is invalid! This may have applications in, say, security where the only goal is to thwart an attacker, but certainly no programmer would want their program to run forever when they accidentally use a number as a function.

Instead, we can separate our two notions of "definedness" on our computations: one to interpret divergence, which will reproduce ordinary domain theory, and another for type errors, which will be used in our interpretation of ep pairs and the gradual guarantee. The main difference is that the type error denotes a finitely observable notion of undefined, whereas we can never know for sure that a black-box program has diverged. This means our domains and predomains will have two orderings on them, the "divergence" ordering will be notated by $\leq_{\text{div}}$ and the "error" ordering will be notated by $\sqsubseteq_{\text{err}}$. Domains will then be equipped with two least elements, Ω will be least according to $\leq_{\text{div}}$ and represents diverging computations, whereas $\mho$ will be least according to $\sqsubseteq_{\text{err}}$ and represents erroring computations. Finally while certain limits of chains will exist with respect to $\leq_{\text{div}}$, with respect to $\sqsubseteq_{\text{err}}$ the domain is merely a poset.

First, we define predomains, which serve as the interpretation of types of *values*, and continuous functions will interpret those "pure" terms that never error or diverge.

Definition 5.1. Predomains, Continuous Functions

(1) A predomain X is a triple $(|X|, \leq_X, \sqsubseteq_X)$ such that $\leq_X, \sqsubseteq_X)$ are partial orders on $|X|$, $\leq_X$ is closed under directed limits, and $\sqsubseteq_X$ admits $\leq_X$-induction: for any directed $D, E \subseteq X$, we have $\bigvee D \sqsubseteq_X \bigvee E$ if for any $d \in D, e \in E$, there exists an $d' \in D$ $e' \in E$ with $d \leq_X d', e \leq_X e'$ such that $d \sqsubseteq_X e$.

(2) A continuous function $f : X \to X'$ where X, X' are predomains is a function $f : |X| \to |X'|$ of underlying sets so that $f : (|X|, \leq_X) \to (|X'|, \leq_{X'})$ is monotone and preserves directed limits and $f : (|X|, \sqsubseteq_X) \to (|X'|, \sqsubseteq_{X'})$ is monotone.

This ensures, $(|X|, \leq_X)$ is a dcpo and $(|X|, \sqsubseteq_X)$ is a poset. We call $\leq_X$ the *divergence order* and we call $\sqsubseteq_X$ the *error order*. We denote by PreDom the category of predomains and continuous functions, with the obvious identities and composition. Then we have three notions of domain of interest, one for errors, one for divergence and a third for their combination.

Definition 5.2. Domains

(1) An error domain is a predomain X with a $\sqsubseteq_X$-least element
(2) A divergence domain is a predomain X with a $\leq_X$-least element.
(3) A domain is a predomain X that is both an error domain and a divergence domain.

and we have three corresponding monads that allow us to construct domains from predomains by freely adding least elements: $X_{\mho}, X_{\Omega}, X_{\Omega,\mho}$. These monads give us three kleisli categories, $\mathrm{PreDom}_{\mho}, \mathrm{PreDom}_{\Omega}, \mathrm{PreDom}_{\Omega,\mho}$, that each serve an important purpose in our semantics. The category PreDom_{Ω} of possibly diverging functions, which is CPPO-enriched using $\leq_{\mathrm{div}}$, is the setting for the classical domain theoretic aspects, such as solutions of recursive domain equations. The category $\mathrm{PreDom}_{\mho}$ of possibly erroring functions, poset-enriched using $\sqsubseteq_{\mathrm{err}}$, is the setting for interpreting gradual types as ep pairs. Finally, the category $\mathrm{PreDom}_{\Omega,\mho}$ of effectful functions is the target of our denotational semantics of the core language.

5.2 Semantics and Adequacy

First, we give a fairly traditional denotational semantics of our core language in $\mathrm{PreDom}_{\Omega,\mho}$. Since the core language is not gradually typed, and instead simply has a recursive type $?$, this semantics all follows standard constructions , the full semantics is in the appendix. The sums and products are interpreted as coproducts and products in PreDom, the function type is interpreted as the predomain of possibly erroring, diverging functions, and the partial product type $\prod_{\mathbf{i}\in\mathbf{I}}\mathbf{A}_i$ is interpreted as the subdomain of the product of $[\![\mathbf{A}_i]\!]_{\mho}$ consisting of products with finite support. Finally, the dynamic type is interpreted as a minimal invariant ([12, 20])

$$[\![?]\!] \cong [\![1 + \mathbb{N} + (? + ?) + (? \times ?) + (? \to ?) + \prod_{-\in\mathfrak{F}} ?]\!].$$

We can verify that the conditions described in [28] are satsified by PreDom_{Ω} and so we know that such a solution exists. In this construction we use the category of embedding-projection pairs in PreDom_{Ω} with the $\leq_{\mathrm{div}}$ ordering, but other than establishing existence and basic properties of $[\![?]\!]$, these ep pairs play no role in the rest of our development.

The crucial link between the denotational semantics and operational semantics is our adequacy theorem, which tells us that the semantics is sound for reasoning about errors, divergence and termination in the core language, which we prove using a logical relation following [20]:

Theorem 5.3 (Adequacy of the Denotational Semantics). *If $\cdot \vdash \mathbf{t} : \mathbf{A}$, then if $[\![\mathbf{t}]\!] = \Omega$, then $\mathbf{t}$ diverges, if $[\![\mathbf{t}]\!] = \mho$, then $\mathbf{t} \longmapsto^* \mho$ and if $[\![\mathbf{t}]\!] = \lfloor x \rfloor$, then there exists $\mathbf{v}$, with $\mathbf{t} \longmapsto^* \mathbf{v}$.*

With this in hand, we can easily transport denotational properties to contextual approximation properties:

Corollary 5.4. *Soundness of Error Order for Error Approximation*
If $\Gamma \vdash \mathbf{t} : \mathbf{A}, \Gamma \vdash \mathbf{u} : \mathbf{A}$ and $[\![\mathbf{t}]\!] \sqsubseteq_{[\![\Gamma \to \mathbf{A}_{\Omega,\mho}]\!]} [\![\mathbf{u}]\!]$, then $\mathbf{t} \sqsubseteq^{ctx}_{err} \mathbf{u}$.

5.3 Soundness of Type Precision

Adequacy in hand, we can now prove the soundness theorems of the previous section by proving their semantic analogues, which are much simpler because $\sqsubseteq_A$ is easier to reason with.

First, we need to define what our semantic ep pairs are, so here we give three definitions that turn out to be equivalent. Notably, we show that embeddings are always pure functions.

Definition 5.5 (Error Embedding-projection pair). We define an embedding-projection pair $e, p : X \triangleleft Y$ where X, Y are predomains to be given by any of the following equivalent data:

(1) Two effectful functions $e : X \to_{\Omega,\mho} Y, p : Y \to_{\Omega,\mho} X$ (arrows in $\text{PreDom}_{\Omega,\mho}$) satisfying $p \circ e = \text{id}_{X_{\Omega,\mho}}, e \circ p \sqsubseteq_{Y \to_{\Omega,\mho} Y} \text{id}_{Y_{\Omega,\mho}}$.
(2) Two possibly erroring function $e : X \to_{\mho} Y, p : Y \to_{\mho} X$ (arrows in $\text{PreDom}_{\mho}$) satisfying $p \circ e = \text{id}_{X_{\mho}}, e \circ p \sqsubseteq_{Y \to_{\mho} Y} \text{id}_{Y_{\mho}}$.
(3) Two functions, a pure function $e : X \to Y$ and a possibly erroring function $p : Y \to_{\mho} X$ satisfying $p \circ e' = \text{id}_{X_{\mho}}, e' \circ p \sqsubseteq_{Y \to_{\mho} Y} \text{id}_{Y_{\mho}}$. where $e' = \lfloor \cdot \rfloor \circ e$.

The first definition is clearly sound for the definition of embedding-projection pair in terms of contextual error approximation in the previous section. The second definition is the most categorically natural, it is just the definition of ep pair in $\text{PreDom}_{\mho}$. The third definition is convenient for some technical purposes and validates the intuition that an embedding, at least in a first-order sense, always succeeds.

Next, we show that the elaborations of a type precision judgment $[\![d^e]\!], [\![d^p]\!]$ form an ep pair by showing that all the rules denote operations that construct ep pairs. In particular, any mixed-arity functor $F : (\text{PreDom}_{\mho}^{op})^n \times \text{PreDom}_{\mho}^m \to \text{PreDom}_{\mho}$ acts on ep pairs *covariantly*, a fact that is vital for the construction of recursive domains, and justifies all of the congruence rules of our language.

Lemma 5.6 (EP Pair Constructions). (1) *$id_X, id_{X_{\Omega,\mho}} : X \triangleleft X$*
(2) *If $e_1, p_1 : X \triangleleft Y, e_2, p_2 : Y \triangleleft Z$, then $e_2 \circ e_1, p_1 \circ p_2 : X \triangleleft Z$*
(3) *For any X there is a unique ep pair $e, \mho : 0 \triangleleft X$.*
(4) *For any functor $F : (PreDom_{\mho}^{op})^n \times PreDom_{\mho}^m \to PreDom_{\mho}$, if all $(e_i, p_i) : X_i \triangleleft Y_i$, then*

$$(F(p_1, \ldots, p_n, e_{n+1}, \ldots, e_{n+m}), F(e_1, \ldots, e_n, p_{n+1}, \ldots, p_{n+m})) : F(X_1, \ldots, X_{n+m}) \triangleleft F(Y_1, \ldots, Y_{n+m})$$

Then we can prove that our precision judgments are ep pairs by verifying that each of the syntactic constructions is interpreted as the corresponding semantic instruction in the denotational semantics.

Lemma 5.7 (Type Precision Judgments, Types are EP Pairs). *For any derivation $\mathsf{d} :: \mathsf{A} \sqsubseteq \mathsf{B}$, $[\![d^e]\!], [\![d^p]\!] : [\![A]\!] \triangleleft [\![B]\!]$. In particular, $[\![A^e]\!], [\![A^e]\!] : [\![A]\!] \triangleleft [\![?]\!]$.*

Next, we prove soundness of type precision with respect to *semantic* type precision.

Lemma 5.8 (Type Precision Judgments are EP Pair Factorizations). *For any $\mathsf{d} :: \mathsf{A} \sqsubseteq \mathsf{B}$ then $[\![d^e]\!], [\![d^p]\!] : [\![A]\!] \triangleleft [\![B]\!]$ satisfies $[\![A^e]\!] = [\![B^e]\!] \circ [\![d^e]\!]$ and $[\![A^p]\!] = [\![d^p]\!] \circ [\![B^p]\!]$.*

Proof. Proven by structural induction on the type precision judgment d and a careful analysis of the definition of A^e, A^p. The full details are in the appendix, but we show an illustrative case, that shows the key use of *functoriality* in the proof. Consider $\mathsf{d} \to \mathsf{d}' :: \mathsf{A} \to \mathsf{B} \sqsubseteq \mathsf{A}' \to \mathsf{B}'$. We want to show that

$$[\![\mathsf{Imp}(A \to B)]\!] = [\![d \to d']\!]; \mathsf{Imp}(A' \to B')$$

By definition, $\mathsf{Imp}(A' \to B') = (\mathsf{Imp}(A') \to \mathsf{Imp}(B'));\mathsf{Tag}_{? \to ?}$. Then by functoriality of $\to$, we have

$$[\![d \to d']\!]; [\![\mathsf{Imp}(A') \to \mathsf{Imp}(B')]\!] = [\![(d;\mathsf{Imp}(A')) \to (d';\mathsf{Imp}(B'))]\!]$$

$$\frac{\Gamma \sqsubseteq \Gamma' \qquad A \sqsubseteq A'}{\Gamma, x : A \vdash x : A \sqsubseteq \Gamma', x : A' \vdash x : A'} \qquad \frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : ?}{\Gamma \vdash \Uparrow^{?}_{A} (t) : ? \sqsubseteq \Gamma' \vdash t' : ?} \qquad \frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A'}{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash \Uparrow^{?}_{A'} (t') : ?}$$

$$\frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : ? \qquad A \sqsubseteq A'}{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash \Downarrow^{?}_{A'} (t') : A'} \qquad \frac{\Gamma \vdash t : A_1 \sqsubseteq \Gamma' \vdash t' : ? \qquad A_2 \sqsubseteq A_1}{\Gamma \vdash \Downarrow^{?}_{A_2} (t) : A_2 \sqsubseteq \Gamma' \vdash t' : ?}$$

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

$$\frac{\Gamma \vdash t : A_t \sqsubseteq \Gamma' \vdash t' : A'_t \qquad \Gamma \vdash u : A'_u \sqsubseteq \Gamma' \vdash u' : A_u \qquad B \sqsubseteq B'}{\Gamma \vdash (t @ A_t \Rightarrow A_u \to B)\, u : B \sqsubseteq \Gamma' \vdash (t' @ A'_t \Rightarrow A'_u \to B')\, u' : B'}$$

Fig. 11. Term Precision (Fragment)

And by inductive hypothesis $[\![d;\mathrm{Imp}(A')]\!] = [\![\mathrm{Imp}(A)]\!]$, $[\![d';\mathrm{Imp}(B')]\!] = [\![\mathrm{Imp}(B)]\!]$, so the result follows by definition of $\mathrm{Imp}(A \to B)$. □

6 GRADUAL GUARANTEE

Now we consider the gradual guarantee, a soundness principle for gradually typed languages introduced in [24]. We view the gradual guarantee as being a natural extension of the projection property of ep pairs $e(p(x)) \sqsubseteq_{\mathrm{err}} x$ to more general programs. As described in §3.1, type precision naturally arises when a programmer refines an api, changing from fully dynamic to more and more precise static specifications. The purpose of gradual typing is to make this process as easy as possible for the programmer, in particular it should be easy to reason about how the behavior of the program changes when making a type more or less precise: they only affect whether or not certain runtime type checks are performed. This is exactly what the *gradual guarantee* ensures.

Term Precision. To formalize the gradual guarantee, following [24] we introduce *term* precision, a representative subset of the rules are in Figure 11. The judgment $\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A'$ can be read as "the derivation for t on is more precise than the derivation for t'" should be read as relating a (fragment of a) program at two points in the process of development. We can think of starting with the term t' and making some types stricter to get t or starting with t and weakening some types to get t'. In particular the *interface* for t, captured by the types Γ, A should be at least as strict as the interface Γ', A' for t' and this invariant ($\Gamma \sqsubseteq \Gamma', A \sqsubseteq A'$) is maintained by the rules. While many, the rules follow a simple pattern: $t \sqsubseteq t'$ when every cast in t is at more precise types than the corresponding cast in t'. Accordingly, the introduction rules are just congruences and the elimination rules have side-conditions that type precision holds on the relevant types.

Heterogeneous Contextual Approximation. The most natural interpretation of a type precision judgment $\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A'$ is given by contextual error approximation, but as defined that only makes sense when $\Gamma = \Gamma'$ and $A = A'$. Nevertheless, in this specific case, this implies the original definition of the gradual guarantee in [24].

To compare terms with different typing, we note that because any type can be cast to any other, we can define a *heterogenous* context-plugging operation, that inserts casts as needed, and with it a notion of error approximation.

Definition 6.1. Heterogeneous Context Plugging, Approximation
Given $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ and $C : (x_1 : B_1, \ldots, x_n : B_n \vdash B) \Rightarrow (\Gamma' \vdash B')$, we define the

heterogenous plugging

$$C\langle t\rangle \overset{\text{def}}{=} C[\text{let } x_1 = (A_1 \Leftarrow B_1)x_1 \cdots \text{ in } (B' \Leftarrow A)t]$$

We say that $\Gamma \vdash t : A$ approximates $\Gamma' \vdash t' : A'$, written $t \sqsubseteq^{hctx}_{err} t'$ when Γ, Γ' have the same set of variables and for any context $C : (\Gamma'' \vdash A'') \Rightarrow (\cdot \vdash 1)$, we have $C\langle t\rangle \sqsubseteq_{err} C\langle t'\rangle$.

Then, as with type precision, we can get an equivalent definition only using embeddings and projections instead of quantifying over arbitrary casts. At first glance, there seem to be many ways to do this: we can insert either upcasts or downcasts on the inputs and the outputs, which makes for four combinations. Fortunately, all of these definitions coincide with the one in terms of heterogeneous approximation.

Definition 6.2. Semantic Term Precision If $\Gamma \vdash t : A, \Gamma' \vdash t' : A'$, where $d_\Gamma :: \Gamma \sqsubseteq \Gamma', d_A :: A \sqsubseteq A'$, we say t *approximates* u, written $t \sqsubseteq^{sem} t'$ when any of the following hold:

(1) $t \sqsubseteq^{hctx}_{err} t'$
(2) For any $C : (\Gamma' \vdash A') \Rightarrow (\cdot \vdash 1)$, $C\langle t\rangle \sqsubseteq_{err} C[t']$.
(3) For any $C : (\Gamma \vdash A') \Rightarrow (\cdot \vdash 1)$, $C\langle t\rangle \sqsubseteq_{err} C\langle t'\rangle$.
(4) For any $C : (\Gamma' \vdash A) \Rightarrow (\cdot \vdash 1)$, $C\langle t\rangle \sqsubseteq_{err} C\langle t'\rangle$.
(5) For any $C : (\Gamma \vdash A) \Rightarrow (\cdot \vdash 1)$, $C[t] \sqsubseteq_{err} C\langle t'\rangle$.

Then the gradual guarantee is exactly soundness of term precision with respect to semantic term precision.

Denotational Term Precision and Soundness. To prove the gradual guarantee holds, we first develop a simpler denotational counterpart to semantic term precision.

Given $\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A'$, morally we would like to compare them using $\sqsubseteq_{err}$ pointwise, because this semantically captures the error approximation, but we can't directly do this because they are at different types. However, we know that $\Gamma \sqsubseteq \Gamma'$ and $A \sqsubseteq A'$, so we have embeddings and projections for each, so we have in fact many ways to coerce them to have the same type. Fortunately all combinations result in equivalent notions of approximation:

Lemma 6.3. *For any* $\Gamma \vdash t : A, \Gamma' \vdash t' : A'$, *if* $\Gamma \sqsubseteq \Gamma', A \sqsubseteq A'$ *then the following are equivalent:*

(1) $[\![(A \sqsubseteq A')^{e}]\!] \circ [\![t]\!] \sqsubseteq_{err} [\![t']\!] \circ [\![(\Gamma \sqsubseteq \Gamma')^{e}]\!]$.
(2) $[\![t]\!] \circ [\![(\Gamma \sqsubseteq \Gamma')^{p}]\!] \sqsubseteq_{err} [\![(A \sqsubseteq A')^{p}]\!] \circ [\![t']\!]$.
(3) $[\![(A \sqsubseteq A')^{e}]\!] \circ [\![t]\!] \circ [\![(\Gamma \sqsubseteq \Gamma')^{p}]\!] \sqsubseteq_{err} [\![t']\!]$.
(4) $[\![t]\!] \sqsubseteq_{err} [\![(A \sqsubseteq A')^{p}]\!] \circ [\![t']\!] \circ [\![(\Gamma \sqsubseteq \Gamma')^{e}]\!]$.

Where $\sqsubseteq_{err}$ *is the pointwise ordering:* $f \sqsubseteq_{err} g$ iff $\forall x. f(x) \sqsubseteq_{err} g(x)$.

Proof. The equivalence of the first two is standard 2-category theory. 3 can be derived from 2 and 4 from 1 by using the elementwise definition of an adjunction. For predomains X, Y and an ep pair form X to Y and any $x \in X, y \in Y$, $e(x) \sqsubseteq_Y y$ if and only if $x \sqsubseteq_X p(y)$. □

We could in principle, now prove the soundness of term precision with respect to this semantics, but the proof would be somewhat tedious because we would have to deconstruct the definitions of the ep pairs in each case. While, the definition of the ep pairs is compositional, the notion of cross-type ordering derived from ep pairs is not defined compositionally. Instead, we define a *logical relation* definition of cross-type $\sqsubseteq_{err}$ indexed by precision derivations, that is for each $d :: A \sqsubseteq B$ a *relation* between A and B. Intuitively, a "cross-type" $\sqsubseteq_{err}$ is one that interacts well with $\sqsubseteq_{err}$ on each side, monotonically on one side and anti-tonically in the other[4]:

[4]In fact in the appendix we also use that the relations we define are *admissible*, but we ignore this here for simplicity.

Definition 6.4. Monotone Relation A monotone relation *from* a predomain X, *to* a predomain Y, written $R : X \nrightarrow Y$, is a subset $R \subset |X| \times |Y|$ of the underlying sets that is downward-closed in $\sqsubseteq_X$ and upper-closed in $\sqsubseteq_Y$, meaning if $(x, y) \in R, x' \sqsubseteq_X x, y \sqsubseteq_Y y'$ then $(x', y') \in R$. We write membership $R \vDash x \sqsubseteq y$ as evocative notation for $(x, y) \in R$.

To relate computations, we need to lift relations from predomains to domans. To turn our ep pairs into relations as above, we use the *pullback* and *pushforward*:

Definition 6.5. Lift, Pullback, Pushforward

(1) For any monotone relation $R : X \nrightarrow Y$, we can define the lift $R_{\Omega,\mho} : X_{\Omega,\mho} \nrightarrow Y_{\Omega,\mho}$ as the least relation satisfying $R_{\Omega,\mho} \vDash \mho \sqsubseteq y$ and $R_{\Omega,\mho} \vDash \Omega \sqsubseteq \Omega$ and $R_{\Omega,\mho} \vDash \lfloor x \rfloor \sqsubseteq \lfloor y \rfloor$
(2) For any $f : X \to Y$ we can define the *pullback* $f^* : X \nrightarrow Y$ by $f^* \vDash x \sqsubseteq y = f(x) \sqsubseteq_Y y$.
(3) For any $g : X \to Y_\mho$ we can define the *pushforward* $g_! : Y \nrightarrow X$ by $g_! \vDash y \sqsubseteq x = \lfloor x \rfloor \sqsubseteq_{Y_\mho} g(y)$.

Then the non-compositional definition of a cross-type $\sqsubseteq_{\text{err}}$ given in the proof above is given equivalently by the pullback of the embedding, or pushforward of the projection: $\mathsf{d} \vDash a \sqsubseteq b$ if and only if $\mathsf{d}^{\mathsf{e}*} \vDash a \sqsubseteq b$ if and only if $\mathsf{d}^{\mathsf{p}}_{!} \vDash a \sqsubseteq b$. Which gives us a relational formulation of term precision, clearly equivalent to the above definitions:

$$\forall \Gamma \sqsubseteq \Gamma' \vDash \gamma \sqsubseteq \gamma.\ (\mathsf{A} \sqsubseteq \mathsf{A}')_{\Omega,\mho} \vDash [\![\mathsf{t}]\!](\gamma) \sqsubseteq [\![\mathsf{u}]\!](\gamma')$$

Fortunately, we can *prove* that the non-compositional definition using pullbacks/pushforwards is equivalent to the obvious compositional logical relation because we have constructed our ep pairs in such a categorically natural way: by compositions, identities and application of functors, all of which commute with taking pushforwards/pullbacks. That is, for instance we have that

$$(\mathsf{d} \to \mathsf{d}')^{\mathsf{e}*} \vDash f \sqsubseteq g \text{ iff } \forall \mathsf{d}^{\mathsf{e}*} \vDash x \sqsubseteq y.\ \mathsf{d}'^{\mathsf{e}*}{}_{\Omega,\mho} \vDash f(x) \sqsubseteq g(y)$$

so in this case we can have our cake and eat it too: the simple, non-compositional definition and the convenient compositional definition coincide.

The logical relation, indexed by precision judgments, is presented in Figure 12. It defines for any $\mathsf{d} :: \mathsf{A} \sqsubseteq \mathsf{B}$, a monotone relation $[\![\mathsf{A}]\!] \nrightarrow [\![\mathsf{B}]\!]$. The cases for tags $\mathsf{Tag}_{\mathsf{G}}$, $\mathsf{PrecGrad}\langle\rho\rangle$ are essentially defined by unwinding the definition of $[\![\mathsf{Tag}_{\mathsf{G}}{}^{\mathsf{p}}]\!]_!$, $[\![\mathsf{PrecGrad}\langle\rho\rangle^{\mathsf{p}}]\!]_!$. Identity derivations $\mathsf{A} \sqsubseteq \mathsf{A}$ are interpreted as the error ordering $\sqsubseteq_{[\![\mathsf{A}]\!]}$, and transitivity is interpreted as the composition of relations[5]. The $0 \sqsubseteq \mathsf{A}$ case is the empty relation, and the congruence cases are all the natural actions of the functors on relations.

By using a logical relation, all of the cases for type precision that *don't* use casts follow by basic structural properties. To prove the case for rules that *do* involve casts, we need the following lemma:

Lemma 6.6. *Cast Compatibility If* $\mathsf{d}_{\mathsf{A}} :: \mathsf{A} \sqsubseteq \mathsf{A}', \mathsf{d}_{\mathsf{B}} :: \mathsf{B} \sqsubseteq \mathsf{B}'$, *then for any* $\mathsf{d}_{\mathsf{A}} \vDash a \sqsubseteq a'$, $\mathsf{d}_{\mathsf{B}\,\Omega,\mho} \vDash [\![(\mathsf{A} \Rightarrow \mathsf{B})]\!] a \sqsubseteq [\![(\mathsf{A}' \Rightarrow \mathsf{B}')]\!] a'$.

This lemma relates the logical relation to the casts, and we prove it by showing that the logical relation is equivalent to the definition in terms of pullbacks of the embeddings/pushforwards of the projections. To do this, we observe the following universal property of the pullback (there is a dual property for pushforward):

Lemma 6.7. *Pullback Universal Property For any continuous function* $f : X \to Y$, *the pullback* f^* *is the unique relation* R *such that*

$$R \vDash x \sqsubseteq y \implies f(x) \sqsubseteq_Y y \quad \textit{and} \quad x \sqsubseteq_X x' \implies R \vDash x \sqsubseteq f(x')$$

[5] A composite of admissible relations is not always admissible, but we show all composites we construct are admissible.

$$\begin{array}{rcl@{\qquad}rcl}
\mathrm{id}_{\mathrm{A}} \vDash x \sqsubseteq y & \overset{\text{def}}{=} & x \sqsubseteq_{[\![\mathrm{A}]\!]} y & \mathrm{d}_1 + \mathrm{d}_2 \vDash \sigma_i x \sqsubseteq \sigma_j y & \overset{\text{def}}{=} & i = j \wedge \mathrm{d}_i \vDash x \sqsubseteq y \\
\mathrm{d};\mathrm{d}' \vDash x \sqsubseteq z & \overset{\text{def}}{=} & \exists y. \mathrm{d} \vDash x \sqsubseteq y \wedge \mathrm{d}' \vDash y \sqsubseteq z & \mathrm{d}_1 \times \mathrm{d}_2 \vDash x \sqsubseteq y & \overset{\text{def}}{=} & \forall i \in \{1,2\} \mathrm{d}_i \vDash \pi_i x \sqsubseteq \pi_i y \\
\mathrm{Tag}_{\mathrm{G}} \vDash x \sqsubseteq y & \overset{\text{def}}{=} & \exists y'.\, y = \sigma_{\mathrm{G}} y'. x \sqsubseteq_{[\![\mathrm{G}]\!]} y' & \mathrm{d}_1 \to \mathrm{d}_2 \vDash f \sqsubseteq g & \overset{\text{def}}{=} & \forall \mathrm{d}_1 \vDash x \sqsubseteq y.\, \mathrm{d}_{2\Omega,\mho} \vDash f(x) \sqsubseteq g(y) \\
0_{\mathrm{A}} \vDash x \sqsubseteq y & \overset{\text{def}}{=} & \bot & \mathrm{PrecGrad}\langle \rho \rangle \vDash p \sqsubseteq (p', g) & \overset{\text{def}}{=} & p \sqsubseteq_{[\![\{\rho\}]\!]} p'
\end{array}$$

$$\begin{array}{rcl}
\{\eta_1 : \mathrm{d}_1, \ldots, \eta_n : \mathrm{d}_n\} \vDash (x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) & \overset{\text{def}}{=} & \forall i \in 1, \ldots, n.\ \mathrm{d}_i \vDash x_i \sqsubseteq y_i \\
\{\eta_1 : \mathrm{d}_1, \ldots, \eta_n : \mathrm{d}_n, ?\} \vDash (p, g) \sqsubseteq (p', g') & \overset{\text{def}}{=} & g \sqsubseteq_{[\![\prod_{\mathfrak{F} \setminus \{\eta_1, \ldots, \eta_n\}} ?]\!]} g' \\
& & \quad \wedge \{\eta_1 : \mathrm{d}_1, \ldots, \eta_n : \mathrm{d}_n\} \vDash p \sqsubseteq p' \\
\mathrm{Width}_?\langle \rho; \eta_1, \ldots, \eta_n \rangle \vDash ((p_\rho, p_\eta), g) \sqsubseteq (p'_\rho, g') & \overset{\text{def}}{=} & \{\rho\} \vDash p_\rho \sqsubseteq p'_\rho \\
& & \wedge \prod_{\mathfrak{F} - \rho, \eta_1, \ldots, \eta_n} ? \vDash g \sqsubseteq g|_{\mathfrak{F} - \rho, \eta_1, \ldots, \eta_n} \\
& & \wedge \forall i \in 1, \ldots, n.\ \pi_i g' = \lfloor x_{\eta_i} \rfloor \wedge \pi_{\eta_i} g \sqsubseteq_{[\![?]\!]} x_{\eta_i}
\end{array}$$

Fig. 12. Gradual Guarantee Logical Relation

Then we show the logical relation is the pullback by showing that all of the constructions we use preserve these properties. For our connectives this means showing that their relational action is functorial in a certain sense:

Definition 6.8 (Relational Functor). A mixed-variance functor $F : (\mathrm{PreDom}_{\mho}^{op})^n \times \mathrm{PreDom}_{\mho}^m \to \mathrm{PreDom}_{\mho}$ extends to a relational functor if there is an action on relations:

$$F : ((W_1 \twoheadrightarrow X_1) \times \cdots \times (Y_1 \twoheadrightarrow Z_1) \times \cdots) \to F(W_1, \ldots, W_n, Y_1, \ldots, Y_m) \twoheadrightarrow F(X_1, \ldots, X_n, Z_1, \ldots, Z_m)$$

satisfying:

(1) Identity Extension/Normality: $F(\sqsubseteq_{X_1}, \ldots) = \sqsubseteq_{F(X_1, \ldots)}$
(2) Relational functoriality: if $R_i \to R'_i \vDash f_i \sqsubseteq f'_i$ and $S_j \to S'_j \vDash g_j \sqsubseteq g'_j$, then

$$F(R_1, \ldots, S_1, \ldots) \to F(R'_1, \ldots, S'_1, \ldots) \vDash F(f_1, \ldots, g_1, \ldots) \sqsubseteq F(f'_1, \ldots, g'_1, \ldots)$$

Then using the universal properties of pullback and pushforward, we can see that any relational functor F [23], $F(p_!, \ldots, e^*, \ldots) = F(p, \ldots, e, \ldots)^*$ and $F(e^*, \ldots, p_!, \ldots) = F(e, \ldots, p, \ldots)_!$. Then by inductiion we can establish that our two relational semantics coincide:

Lemma 6.9 (Compositional and Non-compositional Relational Semantics Agree). *For any $d :: \mathrm{A} \sqsubseteq \mathrm{B}$, $x \sqsubseteq_d y$ if and only if $[\![d^{\mathrm{e}}]\!](x) \sqsubseteq_{[\![\mathrm{B}]\!]} y$.*

Then we prove the fundamental lemma for the logical relation by induction on term precision derivations, as a consequence the gradual guarantee.

Lemma 6.10. *Fundamental Lemma*
If $\Gamma \vdash \mathrm{t} : \mathrm{A} \sqsubseteq \Gamma' \vdash \mathrm{t}' : \mathrm{A}'$ then $(\Gamma \sqsubseteq \Gamma') \to (\mathrm{A} \sqsubseteq \mathrm{A}') \vDash [\![\mathrm{t}]\!] \sqsubseteq [\![\mathrm{u}]\!]$.

Corollary 6.11. *Gradual Guarantee*
If $\Gamma \vdash \mathrm{t} : \mathrm{A} \sqsubseteq \Gamma' \vdash \mathrm{t}' : \mathrm{A}'$ then $\mathrm{t} \sqsubseteq_{err}^{hctx} \mathrm{t}'$.

In summary, we see that the gradual guarantee arises quite naturally from the ep pair semantics, and recognizing the semantic significance of the construction of the ep pairs enables a canonical proof technique using a logical relation.

7 EXTENDING THE SURFACE LANGUAGE

Since we have a compositional translation of our source language into our core language, we can *extend* the source language with any type as long as we can specify its semantics. That is if we want to add a new type A, we just need to define its core language implementation type A and its ep pair

$A \triangleleft ?$. As a concrete example, consider a procedure that may either return a function or some a number representing error information. This could be represented in a typed language using a sum type $\mathbb{N} + (A \to B)$. Similarly, in a dynamically typed language, the procedure would return either a number or a function, however, since the $+$ must in general be represented by an additional tag, the dynamic representation of $\mathbb{N} + (A \to B)$ would not be compatible with the dynamically typed code. Instead what we want is the *union* of $\mathbb{N}$ and $A \to B$ *as represented as dynamically typed values*, but we would also like to use sums for the typed portion of the code. We can do this if we *redefine* the way that $\mathbb{N} + (A \to B)$ is embedded in the dynamic type:

$$v_e = \lambda x.\, \mathsf{case}\, x\, \mathsf{of}\, v_n.\, \Uparrow^{?}_{\mathbb{N}} (v_n) \mid v_f.\, \Uparrow^{?}_{A \to B} (v_n)$$

with the obvious corresponding projection. We will call this type $\mathbb{N} \cup (A \to B)$. To construct this new type, we can add new syntax and show its justified by ep pairs in the core language, but since our semantics is compositional, we can do better and allow surface language programmers to *define* types like $\mathbb{N} + (A \to B)$ by some language feature. For any embedding-projection pair $t_e : A \to B, t_p : B \to A$, we can define a new type $\mathsf{Cast}\langle v_e, v_p : A \triangleleft B\rangle$ in the surface language that we call a *cast refinement type*. We call B the *refined type* and A the *implementation type*.

The idea is that our type represents some refinement of B, but representing the refined values using the implementation type A. So typed operations on $\mathsf{Cast}\langle v_e, v_p : A \triangleleft B\rangle$ will be performed on A, but at the dynamic type, values will be represented the same way as B, which is reflected in the precision judgment: $\mathsf{Cast}\langle v_e, v_p : A \triangleleft B\rangle \sqsubseteq B$. Note that $\mathsf{Cast}\langle v_e, v_p : A \triangleleft B\rangle$ is not necessarily more or less precise than A, because semantic type precision reflects how values are encoded in the *dynamic* type, not at their precise type. However, since type precision derivations $d :: C \sqsubseteq A$ defines an ep pair into A^+, and $\mathsf{Cast}\langle v_e, v_p : A \triangleleft B\rangle$ is implemented as A^+, we can compose d *on the left*:

$$\frac{A' \sqsubseteq A}{\mathsf{Cast}\langle v'_e, v'_p : A' \triangleleft B\rangle \sqsubseteq \mathsf{Cast}\langle v_e, v_p : A \triangleleft B\rangle}$$

where v'_e, v'_p are the composition of v_e, v_p with the corresponding downcast,upcast induced by $A' \sqsubseteq A$. However, reasoning about equality of arbitrary ep pairs in a precision judgment is clearly too onerous of a demand on our type checker. On the other hand, by our coherence theorem, we know equality of ep pairs induced by precision judgments is trivial! This means we can maintain a simple syntactic system of precision judgments by building in composition with a precision judgment into the cast refinement types. Then our final representation of a cast refinement type is $\mathsf{Cast}\langle d :: A' \sqsubseteq A; v_e, v_p : A \triangleleft B\rangle$ where B is the *refinee*, A is the *basic implementation type* and A' is the *implementation type*. Then the left precision rule defined in Figure 13 only requires a check for syntactic equality of the ep pairs, which can be made trivial using nominal typing. In terms of our example, we recognize that $\mathbb{N} \cup (A \to B)$, for any A, B is more precise than $\mathbb{N} \cup (? \to ?)$, and we represent it as $\mathsf{Cast}\langle \mathbb{N} + (A \to B) \sqsubseteq \mathbb{N} + (? \to ?); v_e, v_p : \mathbb{N} + (? \to ?) \triangleleft ?\rangle$. We can think of this as defining the *connective* $\mathbb{N} \cup (\cdot \to \cdot)$, with the precision derivations instantiating the holes.

To get term constructors for the type we can add in an isomorphism between $\mathsf{Cast}\langle A' \sqsubseteq A; A \triangleleft B\rangle$ and A' (implemented in the core language as the identity), or we can add analogous term constructors for the type based on the implementation type:

$$\frac{\Gamma \vdash t : A' \qquad \Gamma, x_1 : \mathbb{N} \vdash t_1 : B \qquad \Gamma, x_2 : A \to B \vdash t_2 : C}{\Gamma \vdash \mathsf{case}_{\mathbb{N} \cup (A \to B)}\, t\, \mathsf{of}\, x_1.\, t_1 \mid x_2.\, t_2 : C}$$

Because we have a compositional elaboration semantics, the above analysis can be justified by a simple addition to the elaboration procedure. This makes the interpretation of types and type precision is mutually dependent on the interpretation of terms, but it is clear that at every point, the semantics only depends on the semantics of subderivations.

$$\begin{array}{llll} \text{Types} & \mathsf{A} & ::= & \cdots \mid \mathsf{Cast}\langle \mathsf{d} :: \mathsf{A}' \sqsubseteq \mathsf{A}; \mathsf{v}, \mathsf{v}' : \mathsf{A} \triangleleft \mathsf{B}\rangle \text{ where } (\cdot \vdash \mathsf{v} : \mathsf{A} \to \mathsf{B}, \cdot \vdash \mathsf{v}' : \mathsf{B} \to \mathsf{A}, \mathsf{d} :: \mathsf{A}' \sqsubseteq \mathsf{A}) \\ \text{Terms} & \mathsf{t}, \mathsf{u} & ::= & \cdots \mid \mathsf{enc}_{\mathsf{v},\mathsf{v}}(\mathsf{t}) \mid \mathsf{dec}_{\mathsf{v},\mathsf{v}}(\mathsf{t}) \end{array}$$

$$\mathsf{CastTag}\langle \mathsf{v}_e, \mathsf{v}_p\rangle :: \mathsf{Cast}\langle \mathsf{id}_\mathsf{A} :: \mathsf{A} \sqsubseteq \mathsf{A}; \mathsf{v}_e, \mathsf{v}_p : \mathsf{A} \triangleleft \mathsf{B}\rangle \sqsubseteq \mathsf{B}$$

$$\frac{\mathsf{d} :: \mathsf{A}'' \sqsubseteq \mathsf{A}'}{\mathsf{CastLeft}\langle \mathsf{d}, \mathsf{d}_l, \mathsf{d}_r, \mathsf{v}_e, \mathsf{v}_p\rangle :: \mathsf{Cast}\langle \mathsf{d}_l :: \mathsf{A}'' \sqsubseteq \mathsf{A}; \mathsf{v}, \mathsf{v}' : \mathsf{A} \triangleleft \mathsf{B}\rangle \sqsubseteq \mathsf{Cast}\langle \mathsf{d}_r :: \mathsf{A}' \sqsubseteq \mathsf{A}; \mathsf{v}, \mathsf{v}' : \mathsf{A} \triangleleft \mathsf{B}\rangle}$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{A}'}{\Gamma \vdash \mathsf{enc}_{\mathsf{d},\mathsf{v}_e,\mathsf{v}_p}(\mathsf{t}) : \mathsf{Cast}\langle \mathsf{d} :: \mathsf{A}' \sqsubseteq \mathsf{A}; \mathsf{v}_e, \mathsf{v}_p : \mathsf{A} \triangleleft \mathsf{B}\rangle} \qquad \frac{\Gamma \vdash \mathsf{t} : \mathsf{Cast}\langle \mathsf{d} :: \mathsf{A}' \sqsubseteq \mathsf{A}; \mathsf{v}_e, \mathsf{v}_p : \mathsf{A} \triangleleft \mathsf{B}\rangle}{\Gamma \vdash \mathsf{dec}_{\mathsf{d},\mathsf{v}_e,\mathsf{v}_p}(\mathsf{t}) : \mathsf{A}'}$$

$$\begin{array}{rlrl} \mathsf{Cast}\langle \mathsf{d} :: \mathsf{A}' \sqsubseteq \mathsf{A}; \mathsf{v}_e, \mathsf{v}_p : \mathsf{A} \triangleleft \mathsf{B}\rangle^+ & = \mathsf{A}'^+ & \mathsf{CastTag}\langle \mathsf{v}_e, \mathsf{v}_p\rangle^{\mathsf{e}} & = \mathsf{v}_e{}^+\ [\cdot] \\ (\mathsf{enc}_{\mathsf{d},\mathsf{v}_e,\mathsf{v}_p}(\mathsf{t}))^+ & = (\mathsf{t})^+ & \mathsf{CastTag}\langle \mathsf{v}_e, \mathsf{v}_p\rangle^{\mathsf{p}} & = \mathsf{v}_p{}^+\ [\cdot] \\ (\mathsf{dec}_{\mathsf{d},\mathsf{v}_e,\mathsf{v}_p}(\mathsf{t}))^+ & = (\mathsf{t})^+ & \mathsf{CastLeft}\langle \mathsf{d}, \mathsf{d}_l, \mathsf{d}_r, \mathsf{v}_e, \mathsf{v}_p\rangle^{\mathsf{e},\mathsf{p}} & = \mathsf{d}^{\mathsf{e},\mathsf{p}} \end{array}$$

Fig. 13. Adding Cast Refinements

Furthermore, all of the existing soundness theorems remain true *under the assumption* that all cast refinement types are well-behaved in that $\mathsf{v}_e, \mathsf{v}_p$ are ep pairs, and are proved at this level of generality in the appendix. Fortunately, by type soundness for the core language, even invalid ep pairs will still result in well-defined behavior.

While this approach generates many type precision judgments, it would require a great deal more work (an actual program logic) to close the gap on completeness for sytactic type precision. For instance our union type $\mathbb{N} \cup (\mathsf{A} \to \mathsf{B})$ clearly satisfies $\mathbb{N} \sqsubseteq^{\text{sem}} \mathbb{N} \cup (\mathsf{A} \to \mathsf{B})$ and $\mathsf{A} \to \mathsf{B} \sqsubseteq^{\text{sem}} \mathbb{N} \cup (\mathsf{A} \to \mathsf{B})$, but there's no way to know this without inspecting $\mathsf{v}_e, \mathsf{v}_p$.

8 RELATED WORK AND DISCUSSION

Types as Retractions. The interpretation of types as retracts of a single domain originated in [22] 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, for example see Benton [3], (Favonia) et al. [8].

Most directly similar to this work is the unpublished paper [17], which was developed independently from this work. They also advocate for a semantic reading of gradual types as retractions of the dynamic type, following Scott, and our constructions of retractions from gradual types agree on the types we have in common. However their work does not provide our constructive reading of type precision derivations or our definition of semantic type precision. Furthermore, since they use retractions and not ep pairs, they do not provide a semantic understanding of the gradual guarantee.

Many of the properties of our embedding-projection pair semantics of gradual typing are anticipated in Henglein [14] and Thatte [29]. In Henglein [14], they define a language with a notion of *coercion* $A \rightsquigarrow B$ that correspond to general casts, with primitives of tagging $tc! : tc(?, \ldots) \rightsquigarrow ?$ and untagging $tc? : ? \rightsquigarrow tc(?, \ldots)$ for every type constructor "tc". Crucially, they note 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) (Retract property) 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) (Projection property) 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 contextual equivalence and approximation of a standard call-by-value language. Furthermore, in defining our notion of contextual 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 [29], 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 the same as the error ordering in our semantics. However, they do not define our operational formulation of the ordering as contextual approximation, linked to the denotational definition by the adequacy result. They also do not define anything analogous to our more general logical relation indexed by type precision judgments, nor do they prove that any casts form embedding-projection pairs with respect to this ordering.

Semantics of Casts. Superficially similar to the embedding-projection pair semantics is the threesome casts of [27].

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. EP pair semantics does not directly provide an efficient implementation, in fact it is one of the most naïve implementation techniques. Instead, ep pairs provide a simple specification for the cast language that has useful meta-theoretic properties. It may be advantageous to use ep pair semantics for proving properties like the gradual guarantee and then prove a simulation between a threesome cast implementation and the ep pair "specification".

The *partial type equivalences* (peqs) of [6] are also similar to embedding-projection pairs, instead of requiring the retraction property $p \circ e = \mathrm{id}$, they require the projection property in both directions $p \circ e \leq \mathrm{id}$ and $e \circ p \leq \mathrm{id}$. However, peqs are a semantics for *general casts*, whereas ep pairs are very special casts from which all others can be defined. Given an ep pair semantics for gradual types, the casts $A \Rightarrow B$ and $B \Rightarrow A$ do form a partial type equivalence as a consequence of the projection properties of the ep pairs since we have $p_A e_B p_B e_A \sqsubseteq p_A e_A \sqsubseteq \mathrm{id}$ and vice-versa. As a basis for a semantics of gradual typing, peqs have the same problems as general casts, the compostition of two peqs is a peq, but it is usually stronger than the intended cast. Interesting future work would be to reformulate their library based on ep pairs rather than peqs, which would allow for a realization of the cast refinement types presented in §**??**.

Pairs of Projections. A large influence on our analysis is [11] which decomposes contracts in untyped languages as a pair of "projections", i.e., functions $c : ? \to ?$ satisfying $c \sqsubseteq_{? \to ?} \mathrm{id}$. There is a close relationship between such projections and ep pairs, for any ep pair $e, p : A \triangleleft B$, $e \circ p : B \to B$ is a projection. We argue that pairs of projections are better understood as embedding-projection pairs, where the types involved are *semantic* types, for instance partial equivalence relations on untyped values. For instance, consider their illustrative example contract `nat?` $\to$ `nat?` in Scheme. The view of pairs of projections is that this contract decomposes into two components, one that checks only the positive obligations `any` $\to$ `nat?` and one that checks only the negative obligations `nat?` $\to$ `any`. The embedding-projection pair view is that first and foremost, `nat?` $\to$ `nat?` represents a *type*, the partial equivalence relation $\mathbb{N} \to \mathbb{N}$:

$$\mathbb{N} \to \mathbb{N} \vDash f \sim g = \forall \mathbb{N} \vDash n \sim m. \mathbb{N}_{\Omega,\mho} \vDash f(n) \sim f(m)$$

whose values are functions that, when applied to natural number arguments, return natural number outputs (if they return at all), and are considered equal when they have the same behavior when applied to natural number arguments. Then we can show that their projections form an ep pair `nat?` $\to$ `any`, `any` $\to$ `nat?` $: (\mathbb{N} \to \mathbb{N}) \triangleleft ?$, where $?$ is the identity PER. First, `any` $\to$ `nat?` has the semantic type $? \to (\mathbb{N} \to \mathbb{N})$ even though it does no checking on the domain, because the semantic type $\mathbb{N} \to \mathbb{N}$ encodes a restriction on the context not to call the function on anything but numbers anyway. In other words, `any` (the identity function) has the semantic type $\mathbb{N} \to ?$ because $\mathbb{N}$ is a sub-PER of the identity. Similarly we can show that `nat?` $\to$ `any` has the semantic type $(\mathbb{N} \to \mathbb{N}) \to ?$ since its inputs by assumption already return naturals. It seems promising that we can run this connection the other way and extend our language with blame by saying that embeddings always blame the negative party and projections the positive party.

Actually though, ep pairs in a PER semantics are *more general* than pairs of projections because ep pairs don't have to themselves be projections (thought of now as untyped functions). For example, our implementation of "unions" using disjoint sum types on the typed side, changes the representation when the casts are run, so the embedding and projection would not individually be projections, instead we only need that the composite $e \circ p$ is a projection.

Gradualization. The Gradualizer ([4, 5]) and AGT ([13]) both seek to make language design for gradually typed languages more systematic. However, whereas we have focused on developing a semantics of gradual types and identifying compositional principles for constructing gradual types, these works have focused on automatically generating a syntax and operational semantics for a gradual language from an existing typed language. We conjecture that both Gradualizer and AGT satisfy the semantic definitions we have provided here, i.e., that the upcasts and downcasts to the dynamic type do in fact form an embedding-projection pair and that their type precision is sound for semantic type precision.

Empty Type Precision. In §4.3, we showed that empty type 0 is least element of semantic type precision. However, this disagrees with existing work on gradual typing, where $0 \sqsubseteq A$ is not a type precision rule. We consider how to justify this using AGT, and also using blame soundness.

In Abstracting Gradual Typing [13], gradual types represent sets of "concrete" types from some existing statically typed language without errors, via a concretization function $\gamma : \mathrm{GTy} \to \mathcal{P}\mathrm{CTy}$. Then type precision is induced by the subset operation: $A \sqsubseteq B \mathit{iff} \gamma(A) \subseteq \gamma(B)$. If the static language has an empty type 0, then this will not be the most precise type, since for example it will not be more precise than some other base type: $\gamma(0) = \{0\} \not\subseteq \{\mathbb{N}\} = \gamma(\mathbb{N})$. However, we can make a maximally precise type by fiat: add a new type $\perp$ and define $\gamma(\perp) = \emptyset$. Then it seems like this type is a closer analogue of our 0 than 0. However it is not clear to us what the difference is between

$\bot$ and 0 *semantically*, they both seem to be types with no runtime values and thus semantically correspond to our 0.

Another possibility is to use *blame soundness* to understand the rule. In [24] if $A \sqsubseteq B$, than any program that type checks using A should still type check if replaced by B[6]. With respect to this syntactic check, we can easily make $\bot \sqsubseteq A$ be true by having any program using $\bot$ fail to type check, which is essentially true of the AGT $\bot$ above. To understand what the dynamics of this type should be, we consider blame soundness, proposed in [32], which says that a program should type check if it is never blamed by any inserted cast. Then to make sure any use of $\bot$ fails to type check, we could design the dynamic semantics to *always blame both parties*, since then using $\bot$ in positive or negative position would not type check.

Catchable Contract Errors. An obstacle to implementing the ep pair semantics in practice is that implementing runtime type errors as catchable exceptions violates the gradual guarantee. For instance $\mho \sqsubseteq_{\text{err}} 0$, but if a context could catch the error and return 1, then we would have a counterexample to contextual approximation. We could similarly break this if gradual records were implemented in a language where you could check what fields are defined. Not all reasoning is lost in such a semantics, since for example the retraction property is still true, but, in order to interpret the projection property in a meaningful way in an untyped language, one has to restrict to linking with programs that don't catch contract errors. In a language with some typing distinction for effects, like Haskell, this can be codified in a syntactic type: arbitrary Haskell programs can use `error`, but they can only be caught in the `IO` monad.

9 FUTURE WORK

Blame. While we have not discussed blame, the decomposition of every cast into an upcast followed by a downcast allows for an obvious enrichment of the runtime semantics in which errors thrown by a projection blame the "positive" party whereas embeddings blame the "negative" party. This agrees with our analysis of pairs of projections above in §8. However, we have not tried to incorporate this definition into this paper because it would significantly complicate the model. Such a model must take seriously the *generativity* of blame labels, and thus demands a stateful or nominal semantics.

Subtyping. In the future, we hope to develop a sensible semantic theory of subtyping and gradual subtyping, using our current semantics as a starting point since we already have the record types which have interesting subtyping structure. One possibility is to inherit subtyping directly from a core language with subtyping, but this does not recover the notion of gradual subtype used in the literature, for which the dynamic type is a gradual subtype and supertype of every other type (and the judgment is not transitive).

Other Kinds of Graduality. While we have focused on "traditional" gradual typing, where the graduality is on types describing values, we expect that our semantics can be adapted to other notions of gradual typing such as effects [2] or information flow [7]. For example, for gradual effects, effects would likely correspond to monads and the ep pairs would be pairs of morphisms of monads. We conjecture that our semantics of types, type precision and term precision can be given an analogous treatment there.

Other Language Features. While we have considered the basic building blocks of functional programming (sums, products, functions) in this paper, we expect that we can extend this semantics to other types. The most obvious extension would be to extend to *recursive types* $\mu\alpha.\, A$. The core

[6]Note that our surface language was not designed with this property in mind, satisfying it in a trivial sense.

language and denotational semantics should be easy to extend, however it is not clear whether or not $\sqsubseteq_{\text{err}}$-ep pairs as defined in this paper can be defined by recursion, the retraction property is clear but a straightforward induction proof of the projection property fails because Ω, Ω is not an $\sqsubseteq_{\text{err}}$-ep pair. Also, we plan to analyze recent work on parametric polymorphism and gradual typing [1, 16] in terms of ep pairs. These papers use the type precision rule $\forall \alpha.A \sqsubseteq A[?/\alpha]$, which shows that $\forall$ is fundamentally different from types like $\to$ in that it does not have a distinct tag on the dynamic type, instead instantiating first before tagging at the instantiation's tag.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] 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.*
[2] 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.
[3] Nick Benton. 2005. Embedded Interpreters. *Journal of Functional Programming* 15, 04 (2005), 503–542.
[4] 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).*
[5] 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.
[6] Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. 2016. Partial Type Equivalences for Verified Dependent Interoperability *(ICFP 2016).* 298–310.
[7] Tim Disney and Cormac Flanagan. 2011. Gradual information flow typing. In *International Workshop on Scripts to Programs 2011.*
[8] Keun-Bang Hou (Favonia), Nick Benton, and Robert Harper. 2017. Correctness of compiling polymorphism to dynamic typing. *Journal of Functional Programming* 27 (2017).
[9] Matthias Felleisen. 1990. On the expressive power of programming languages. *ESOP'90* (1990).
[10] 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.
[11] Robby Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *International Symposium on Functional and Logic Programming (FLOPS).*
[12] Peter Freyd. 1991. *Algebraically complete categories.* 95–104.
[13] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).*
[14] Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. 22, 3 (1994), 197–230.
[15] Atsushi Igarashi, Peter Thiemann, Vasco Vasconcelos, and Philip Wadler. 2017. Gradual Session Types. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom.*
[16] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom.*
[17] Harley Eades III and Michael Townsend. 2017. The Combination of Dynamic and Static Typing from a Categorical Perspective. http://metatheorem.org/includes/pubs/Grady-Draft.pdf. (2017). Accessed: 2017-7-7.
[18] Anders Kock. 1995. Monads for which structures are adjoint to units. *Journal of Pure and Applied Algebra* 104, 1 (1995), 41 – 59.
[19] 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.
[20] Andrew M. Pitts. 1996. Relational Properties of Domains. *Information and Computation* 127, 2 (1996), 66 – 90.
[21] Gordon Plotkin and Martín Abadi. 1993. A logic for parametric polymorphism. *Typed Lambda Calculi and Applications* (1993), 361–375.

[22] Dana Scott. 1972. Continuous lattices. In *Toposes, algebraic geometry and logic*. 97–136.
[23] Michael Shulman. 2008. Framed bicategories and monoidal fibrations. *Theory and Applications of Categories* 20, 18 (2008), 650–738.
[24] 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)*.
[25] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop (Scheme)*. 81–92.
[26] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (ECOOP)*.
[27] 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.
[28] Michael B Smyth and Gordon D Plotkin. 1982. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.* 11, 4 (1982).
[29] Satish Thatte. 1990. Quasi-static typing. In *ACM Symposium on Principles of Programming Languages (POPL)*. 367–381.
[30] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium (DLS)*. 964–974.
[31] 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*.
[32] Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*. 1–16.

$$
\begin{array}{llcl}
\text{Types} & A, B, C & ::= & ? \mid \mathbb{N} \mid 0 \mid A + B \mid 1 \mid A \times B \mid A \to B \mid \{\rho\} \mid \{\rho, ?\} \\
\text{Names} & \eta & \in & \mathfrak{F} \\
\text{Rows} & \rho & ::= & \eta : A, \dots, \eta : A \\
\text{Environments} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{Terms} & t, u & ::= & x \mid \Uparrow^{?}_{A}(t) \mid \Downarrow^{?}_{A}(t) \mid \mathsf{roll}_{?}\, t \mid \mathsf{unroll}_{?}\, t \mid \mathsf{rec}\, x\, y\, .\, t \mid n \mid \mathsf{add1}\,(t @ A) \mid \mathsf{sub1}\,(t @ A) \\
& & \mid & \mathsf{case}\,(t @ A \Rightarrow 0)\,\mathsf{of}\,() \mid \mathsf{case}\,(t @ A \Rightarrow A_1 + A_2)\,\mathsf{of}\, x_1 .\, t_1 \mid x_2 .\, t_2 \mid \mathsf{inj}_i\, t \mid \langle\rangle \mid \langle t_1, t_2 \rangle \\
& & \mid & \pi_1\,(t @ A \Rightarrow A_1 \times ?) \mid \pi_2\,(t @ A \Rightarrow ? \times A_2) \mid \lambda x .\, t \mid (t @ A_t \Rightarrow A_u \to B)\; u \\
& & \mid & \{\eta_1 \mapsto t_1, \dots, \eta_n \mapsto t_n\} \mid (t @ A \Rightarrow \{\eta : B\}).\eta \\
\text{Values} & v & ::= & v \mid n \mid \mathsf{inj}_i\, v \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid \lambda x .\, t \mid \{\eta_1 \mapsto v_1, \dots, \eta_n \mapsto v_n\}
\end{array}
$$

Fig. 14. Surface Language: Types, Contexts, Terms, Values

$$
\frac{}{\Gamma, x : A \vdash x : A}
\qquad
\frac{\Gamma \vdash t : A}{\Gamma \vdash \Uparrow^{?}_{A}(t) : ?}
\qquad
\frac{\Gamma \vdash t : ?}{\Gamma \vdash \Downarrow^{?}_{A}(t) : A}
$$

$$
\frac{\Gamma \vdash t : 1 + \mathbb{N} + (? + ?) + (? \times ?) + \{?\} + (? \to ?)}{\Gamma \vdash \mathsf{roll}_{?}\, t : ?}
$$

$$
\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{unroll}_{?}\,(t @ A) : 1 + \mathbb{N} + (? + ?) + (? \times ?) + \{?\} + (? \to ?)}
\qquad
\frac{\Gamma, x : A \to B, y : A \vdash t : B}{\Gamma \vdash \mathsf{rec}\, x\, y\, .\, t : A \to B}
$$

$$
\frac{}{\Gamma \vdash n : \mathbb{N}}
\qquad
\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{add1}\,(t @ A) : \mathbb{N}}
\qquad
\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{sub1}\,(t @ A) : 1 + \mathbb{N}}
$$

$$
\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{case}\,(t @ A \Rightarrow 0)\,\mathsf{of}\,() : B}
\qquad
\frac{\Gamma \vdash t : A \qquad \Gamma, x_1 : A_1 \vdash t_1 : B \qquad \Gamma, x_2 : A_2 \vdash t_2 : B}{\Gamma \vdash \mathsf{case}\,(t @ A \Rightarrow A_1 + A_2)\,\mathsf{of}\, x_1 .\, t_1 \mid x_2 .\, t_2 : B}
$$

$$
\frac{\Gamma \vdash t : A_i}{\Gamma \vdash \mathsf{inj}_i\, t : A_1 + A_2}
\qquad
\frac{}{\Gamma \vdash \langle\rangle : 1}
\qquad
\frac{\Gamma \vdash t_1 : A_1 \qquad \Gamma \vdash t_2 : A_2}{\Gamma \vdash \langle t_1, t_2 \rangle : A_1 \times A_2}
$$

$$
\frac{\Gamma \vdash t : A}{\Gamma \vdash \pi_1\,(t @ A \Rightarrow A_1 \times ?) : A_1}
\qquad
\frac{\Gamma \vdash t : A}{\Gamma \vdash \pi_2\,(t @ A \Rightarrow ? \times A_2) : A_2}
$$

$$
\frac{\forall i \in 1 \dots, n.\ \Gamma \vdash t_i : A_i}{\Gamma \vdash \{\eta_1 \mapsto t_1, \dots, \eta_n \mapsto t_n\} : \{\eta_1 : A_1, \dots, \eta_n : A_n\}}
\qquad
\frac{\Gamma \vdash t : A}{\Gamma \vdash (t @ A \Rightarrow \{\eta : B\}).\eta : B}
$$

$$
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x .\, t : A \to B}
\qquad
\frac{\Gamma \vdash t : A_t \qquad \Gamma \vdash u : A_u}{\Gamma \vdash (t @ A_t \Rightarrow A_u \to B)\; u : B}
$$

Fig. 15. Surface Typing Derivations

A APPENDIX

A.1 Extended Figures

A.2 Error Domain Theory

First, we define a notion of domain suitable for interpreting divergence and error orderings.

While in general neither divergence nor error ordering imply the other, we do need that the error ordering for our domains respects the notion of *approximation* provided by the divergence ordering.

$$\begin{array}{llcl}
\text{Type Tags} & G & ::= & \mathbb{N} \mid ?+? \mid 1 \mid ?\times? \mid ?\to? \mid \{?\} \\
\text{Type Precision} & d & ::= & \mathsf{id}_A \mid d;d \mid 0_A \mid \mathsf{Tag}_G \mid d+d \mid d\times d \mid d\to d \\
 & & \mid & \{d\} \mid \mathsf{PrecGrad}\langle\rho\rangle \mid \{d,?\} \mid \mathsf{Width}_?\langle\rho;\eta,\ldots,\eta\rangle \\
\text{Row Precision} & d_\rho & ::= & \eta_1 : d_1,\ldots,\eta_n : d_n \\
\text{Environment Precision} & d_\Gamma & ::= & \cdot \mid d_\Gamma, d
\end{array}$$

$$\frac{}{\mathsf{id}_A :: A \sqsubseteq A} \qquad \frac{d :: A \sqsubseteq B \qquad d' :: B \sqsubseteq C}{d;d' :: A \sqsubseteq C} \qquad \frac{}{\mathsf{Tag}_G :: G \sqsubseteq ?}$$

$$\frac{d :: A \sqsubseteq A' \qquad d' :: B \sqsubseteq B'}{d + d' :: A + B \sqsubseteq A' + B'} \qquad \frac{d :: A \sqsubseteq A' \qquad d' :: B \sqsubseteq B'}{d \times d' :: A \times B \sqsubseteq A' \times B'} \qquad \frac{d :: A \sqsubseteq A' \qquad d' :: B \sqsubseteq B'}{d \to d' :: A \to B \sqsubseteq A' \to B'}$$

$$\frac{}{0_A :: 0 \sqsubseteq A} \qquad \frac{d_\rho :: \rho \sqsubseteq \rho'}{\{d_\rho\} :: \{\rho\} \sqsubseteq \{\rho'\}} \qquad \frac{}{\mathsf{PrecGrad}\langle\rho\rangle :: \{\rho\} \sqsubseteq \{\rho,?\}}$$

$$\frac{d_\rho :: \rho \sqsubseteq \rho'}{\{d_\rho,?\} :: \{\rho,?\} \sqsubseteq \{\rho',?\}} \qquad \frac{}{\mathsf{Width}_?\langle\rho;\eta_1,\ldots,\eta_n\rangle :: \{\rho,\eta_1 : ?,\ldots,\eta_n : ?,?\} \sqsubseteq \{\rho,?\}}$$

$$\frac{\forall i \in \{1,\ldots,n\}.\ d_i :: A_i \sqsubseteq B_i}{\eta_1 : d_1,\ldots,\eta_n : d_n :: \eta_1 : A_1,\ldots,\eta_n : A_n \sqsubseteq \eta_1 : B_1,\ldots,\eta_n : B_n} \qquad \frac{}{\cdot :: \cdot \sqsubseteq \cdot}$$

$$\frac{d :: \Gamma \sqsubseteq \Gamma' \qquad d_x :: A \sqsubseteq A'}{d, x : d_x :: \Gamma, x : A \sqsubseteq \Gamma', x : A'}$$

Fig. 16. Type Precision

$$\begin{array}{rcl}
\mathsf{Imp}(A) & :: & A \sqsubseteq ? \\
\mathsf{Imp}(?) & = & \mathsf{id}_? \\
\mathsf{Imp}(\mathbb{N}) & = & \mathsf{Tag}_{\mathbb{N}} \\
\mathsf{Imp}(1) & = & \mathsf{Tag}_1 \\
\mathsf{Imp}(0) & = & 0_? \\
\mathsf{Imp}(A+B) & = & (\mathsf{Imp}(A) + \mathsf{Imp}(B));\mathsf{Tag}_{?+?} \\
\mathsf{Imp}(A\times B) & = & (\mathsf{Imp}(A) \times \mathsf{Imp}(B));\mathsf{Tag}_{?\times?} \\
\mathsf{Imp}(A\to B) & = & (\mathsf{Imp}(A) \to \mathsf{Imp}(B));\mathsf{Tag}_{?\to?} \\
\mathsf{Imp}(\{\eta_1 : A_1,\ldots,\eta_n : A_n\}) & = & \{\eta_1 : \mathsf{Imp}(A_1),\ldots,\eta_n : \mathsf{Imp}(A_n)\};\mathsf{PrecGrad}\langle\eta_1 : ?,\ldots,\eta_n : ?\rangle; \\
 & & \ \mathsf{Width}_?\langle\cdot;\eta_1,\ldots,\eta_n\rangle;\mathsf{Tag}_{\{?\}} \\
\mathsf{Imp}(\{\eta_1 : A_1,\ldots,\eta_n : A_n,?\}) & = & \{\eta_1 : \mathsf{Imp}(A_1),\ldots,\eta_n : \mathsf{Imp}(A_n),?\};\mathsf{Width}_?\langle\cdot;\eta_1,\ldots,\eta_n\rangle;\mathsf{Tag}_{\{?\}}
\end{array}$$

Fig. 17. Dynamic Type Imprecision

First, we define pre-domains, which serve as the interpretation of types of *values*, and continuous functions will interpret those terms that never error or diverge.

Definition A.1. Pre-Domains, Continuous Functions

(1) A predomain X is a triple $(|X|, \leq_X, \sqsubseteq_X)$ such that $(|X|, \leq_X)$ and $(|X|, \sqsubseteq_X)$ are preorders, $(|X|, \leq_X)$ is closed under directed limits, and that $\sqsubseteq_X$ respects directed limits in $\leq_X$ in that if D, E are directed subsets of X then $\bigvee D \sqsubseteq_X \bigvee E$ if for any $d \in D, e \in E$, there exists an $d' \in D\ e' \in E$ with $d \leq_X d', e \leq_X e'$ such that $d \sqsubseteq_X e$.

This ensures, $(|X|, \leq_X)$ is a dcpo and $(|X|, \sqsubseteq_X)$ is a preorder. We call $\leq_X$ the *divergence ordering* and we call $\sqsubseteq_X$ the *error ordering*.

$$\frac{\Gamma \sqsubseteq \Gamma' \qquad A \sqsubseteq A'}{\Gamma, x : A \vdash x : A \sqsubseteq \Gamma', x : A' \vdash x : A'} \qquad \frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : ?}{\Gamma \vdash \Uparrow^{?}_{A} (t) : ? \sqsubseteq \Gamma' \vdash t' : ?} \qquad \frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A'}{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash \Uparrow^{?}_{A'} (t') : ?}$$

$$\frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : ? \qquad A \sqsubseteq A'}{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash \Downarrow^{?}_{A'} (t') : A'} \qquad \frac{\Gamma \vdash t : A_1 \sqsubseteq \Gamma' \vdash t' : ? \qquad A_2 \sqsubseteq A_1}{\Gamma \vdash \Downarrow^{?}_{A_2} (t) : A_2 \sqsubseteq \Gamma' \vdash t' : ?}$$

$$\frac{\Gamma, x : A \to B, y : A \vdash t : A \to B \sqsubseteq \Gamma', x : A' \to B', y : A' \vdash t' : A' \to B'}{\Gamma \vdash \mathrm{rec}\, x\, y\,.\, t : A \to B \sqsubseteq \Gamma' \vdash \mathrm{rec}\, x\, y\,.\, t' : A' \to B'} \qquad \frac{\Gamma \sqsubseteq \Gamma'}{\Gamma \vdash n : \mathbb{N} \sqsubseteq \Gamma' \vdash n : \mathbb{N}}$$

$$\frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A' \qquad A \sqsubseteq A'}{\Gamma \vdash \mathrm{add1}\,(t \mathbin{@} A) : \mathbb{N} \sqsubseteq \Gamma' \vdash \mathrm{add1}\,(t' \mathbin{@} A') : \mathbb{N}}$$

$$\frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A' \qquad A \sqsubseteq A'}{\Gamma \vdash \mathrm{sub1}\,(t \mathbin{@} A) : 1 + \mathbb{N} \sqsubseteq \Gamma' \vdash \mathrm{sub1}\,(t' \mathbin{@} A') : 1 + \mathbb{N}}$$

$$\frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A' \qquad B \sqsubseteq B'}{\Gamma \vdash \mathrm{case}\,(t \mathbin{@} A \Rightarrow 0)\,\mathrm{of}\,() : B \sqsubseteq \Gamma' \vdash \mathrm{case}\,(t' \mathbin{@} A' \Rightarrow 0)\,\mathrm{of}\,() : B'}$$

$$\frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A' \qquad \Gamma, x_i : A_i \vdash t_i : B \sqsubseteq \Gamma', x_i : A'_i \vdash t'_i : B' \qquad A_i \sqsubseteq A'_i}{\Gamma \vdash \mathrm{case}\,(t \mathbin{@} A \Rightarrow A_1 + A_2)\,\mathrm{of}\, x_1 . t_1 \mid x_2 . t_2 : B \sqsubseteq \Gamma' \vdash \mathrm{case}\,(t' \mathbin{@} A' \Rightarrow A'_1 + A'_2)\,\mathrm{of}\, x_1 . t_1 \mid x_2 . t_2 : B'}$$

$$\frac{\Gamma \vdash t : A_1 \sqsubseteq \Gamma' \vdash t' : A'_1 \qquad A_2 \sqsubseteq A'_2}{\Gamma \vdash \mathrm{inj}_1\, t : A_1 + A_2 \sqsubseteq \Gamma' \vdash \mathrm{inj}_1\, t' : A'_1 + A'_2} \qquad \frac{\Gamma \vdash t : A_2 \sqsubseteq \Gamma' \vdash t' : A'_2 \qquad A_1 \sqsubseteq A'_1}{\Gamma \vdash \mathrm{inj}_2\, t : A_1 + A_2 \sqsubseteq \Gamma' \vdash \mathrm{inj}_2\, t' : A'_1 + A'_2}$$

$$\frac{\Gamma \sqsubseteq \Gamma'}{\Gamma \vdash \langle\rangle : 1 \sqsubseteq \Gamma' \vdash \langle\rangle : 1} \qquad \frac{\Gamma \vdash t_1 : A_1 \sqsubseteq \Gamma' \vdash t'_1 : A'_1 \qquad \Gamma \vdash t_2 : A_2 \sqsubseteq \Gamma' \vdash t'_2 : A'_2}{\Gamma \vdash \langle t_1, t_2 \rangle : A_1 \times A_2 \sqsubseteq \Gamma' \vdash \langle t'_1, t'_2 \rangle : A'_1 \times A'_2}$$

$$\frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A' \qquad A_1 \sqsubseteq A'_1}{\Gamma \vdash \pi_1\,(t \mathbin{@} A \Rightarrow A_1 \times ?) : A_1 \sqsubseteq \Gamma' \vdash \pi_1\,(t' \mathbin{@} A' \Rightarrow A'_1 \times ?) : A'_1}$$

$$\frac{\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A' \qquad A_2 \sqsubseteq A'_2}{\Gamma \vdash \pi_2\,(t \mathbin{@} A \Rightarrow ? \times A_2) : A_2 \sqsubseteq \Gamma' \vdash \pi_2\,(t' \mathbin{@} A' \Rightarrow ? \times A'_2) : A'_2}$$

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

$$\frac{\Gamma \vdash t : A_t \sqsubseteq \Gamma' \vdash t' : A'_t \qquad \Gamma \vdash u : A'_u \sqsubseteq \Gamma' \vdash u' : A_u \qquad B \sqsubseteq B'}{\Gamma \vdash (t \mathbin{@} A_t \Rightarrow A_u \to B)\, u : B \sqsubseteq \Gamma' \vdash (t' \mathbin{@} A'_t \Rightarrow A'_u \to B')\, u' : B'}$$

Fig. 18. Term Precision

(2) A continuous function $f : X \to X'$ where X, X' are error predomains is a function $f : |X| \to |X'|$ of underlying sets so that $f : (|X|, \leq_X) \to (|X'|, \leq_{X'})$ is monotone and preserves directed limits and $f : (|X|, \sqsubseteq_X) \to (|X'|, \sqsubseteq_{X'})$ is monotone. In other words, f is a continuous function of the divergence predomains and a monotone function of the error preorders.

$$
\begin{array}{llcl}
\text{Types} & \mathbf{A}, \mathbf{B} & ::= & ? \mid \mathbb{N} \mid \mathbf{0} \mid \mathbf{A} + \mathbf{B} \mid \mathbf{A} \to \mathbf{B} \mid \bigtimes_{i \in I} A_i (\text{I finite}) \mid \prod_{i \in I} A_i \\
\text{Type Tags} & \epsilon & ::= & \mathbb{N} \mid ? + ? \mid \mathbf{1} \mid ? \times ? \mid ? \to ? \mid \prod_{- \in \mathfrak{F}} ? \\
\text{Terms} & \mathbf{t}, \mathbf{u} & ::= & x \mid \mho \mid \mathbf{roll}_? \, t \mid \mathbf{unroll}_? \, t \mid \mathbf{rec} \, x \, y \, . \, t \mid n \mid \mathbf{sub1}\,(t) \mid \mathbf{case} \, t \, \mathbf{of} \, () \\
 & & \mid & \mathbf{case} \, t \, \mathbf{of} \, x_1 . \, t_1 \mid x_2 . \, t_2 \mid \mathbf{inj}_i \, t \mid \pi_i \, t \mid t.i \lambda x. \, t \mid t \, u \mid \mathbf{let} \, x = t \, \mathbf{in} \, u \\
 & & \mid & \langle t, \ldots, t \rangle_I \mid \{t, \ldots, t\}_I \\
\text{Values} & \mathbf{v} & ::= & x \mid \mathbf{roll}_? \, v \mid n \mid \mathbf{inj}_i \, v \mid \lambda x. \, t \mid \langle v, \ldots, v \rangle_I \mid \{v, \ldots, v\}_I \mid \mathbf{rec} \, x \, y \, . \, t \\
\text{Evaluation Contexts} & \mathbf{E} & ::= & [\cdot] \mid \mathbf{roll}_? \, E \mid \mathbf{unroll}_? \, E \mid \mathbf{sub1}\,(E) \mid \mathbf{case} \, E \, \mathbf{of} \, () \mid \mathbf{case} \, E \, \mathbf{of} \, x_1 . \, t_1 \mid x_2 . \, t_2 \\
 & & \mid & \mathbf{inj}_i \, E \mid \pi_i \, E \mid E \, u \mid v \, E \mid \mathbf{let} \, x = E \, \mathbf{in} \, t \mid \langle v, \ldots, t, \ldots, t \rangle_I \mid \{v, \ldots, t, \ldots, t\}_I \\
\text{Contexts} & \mathbf{C} & ::= & [\cdot] \mid \mathbf{roll}_? \, C \mid \mathbf{unroll}_? \, C \mid \mathbf{rec} \, x \, y \, . \, C \mid \mathbf{sub1}\,(C) \mid \mathbf{case} \, C \, \mathbf{of} \, () \\
 & & \mid & \mathbf{case} \, C \, \mathbf{of} \, x_1 . \, t_1 \mid x_2 . \, t_2 \mid \mathbf{case} \, t \, \mathbf{of} \, x_1 . \, C_1 \mid x_2 . \, t_2 \mid \mathbf{case} \, t \, \mathbf{of} \, x_1 . \, t_1 \mid x_2 . \, C_2 \\
 & & \mid & \mathbf{inj}_i \, C \mid \pi_i \, C \mid C.i \mid \lambda x. \, C \mid C \, u \mid t \, C \mid \mathbf{let} \, x = C \, \mathbf{in} \, u \\
 & & \mid & \mathbf{let} \, x = t \, \mathbf{in} \, C \mid \langle t, \ldots, [\cdot], \ldots, t \rangle_I \mid \{t, \ldots, [\cdot], \ldots, t\}_I
\end{array}
$$

Fig. 19. Core language terms, types

$$E[\cdot : A] : B \overset{\text{def}}{=} x : A \vdash E[x] : B$$

$$C : (\Gamma \vdash A) \Rightarrow (\Gamma' \vdash B) \overset{\text{def}}{=} \forall \Gamma \vdash t : A, \Gamma' \vdash C[t] : B$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : \mathbf{1} + \mathbb{N} + (? + ?) + (? \times ?) + (? \to ?) + \prod_{- \in \mathfrak{F}} ?}{\Gamma \vdash \mathbf{roll}_? \, t : ?}$$

$$\frac{\Gamma \vdash t : ?}{\Gamma \vdash \mathbf{unroll}_? \, t : \mathbf{1} + \mathbb{N} + (? + ?) + (? \times ?) + (? \to ?) + \prod_{- \in \mathfrak{F}} ?} \qquad \frac{}{\Gamma \vdash \mho : A} \qquad \frac{\Gamma \vdash t : A_1}{\Gamma \vdash \mathbf{inj}_1 \, t : A_1 + A_2}$$

$$\frac{\Gamma \vdash t : A_2}{\Gamma \vdash \mathbf{inj}_2 \, t : A_1 + A_2} \qquad \frac{\Gamma \vdash t : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash t_1 : A \qquad \Gamma, x_2 : A_2 \vdash t_2 : A}{\Gamma \vdash \mathbf{case} \, t \, \mathbf{of} \, x_1 . \, t_1 \mid x_2 . \, t_2 : A}$$

$$\frac{\Gamma, x : A_1 \vdash t : A_2}{\Gamma \vdash \lambda x. \, t : A_1 \to A_2} \qquad \frac{\Gamma \vdash t_1 : A_2 \to A \qquad \Gamma \vdash t_2 : A_2}{\Gamma \vdash t_1 \; t_2 : A} \qquad \frac{\Gamma \vdash t_1 : A_1 \qquad \Gamma, x : A_1 \vdash t_2 : A_2}{\Gamma \vdash \mathbf{let} \, x = t_1 \, \mathbf{in} \, t_2 : A_2}$$

$$\frac{\forall 1 \le l < m \le n, i_l \ne i_m \qquad \{i_1, \ldots, i_n\} = I \qquad \forall 1 \le l \le n, \Gamma \vdash t_l : A_{i_l}}{\Gamma \vdash \langle i_1 = t_1, \ldots, i_n = t_n \rangle_I : \bigtimes_{i \in I} A_i} \qquad \frac{\Gamma \vdash t : \bigtimes_{i \in I} A_i \qquad j \in I}{\Gamma \vdash \pi_j \, t : A_j}$$

$$\frac{\forall 1 \le l < m \le n, i_l \ne i_m \qquad \{i_1, \ldots, i_n\} \subseteq I \qquad \forall 1 \le l \le n, \Gamma \vdash t_l : A_{i_l}}{\Gamma \vdash \{i_1 = t_1, \ldots, i_n = t_n\}_I : \prod_{i \in I} A_i} \qquad \frac{\Gamma \vdash t : \prod_{i \in I} A_i \qquad j \in I}{\Gamma \vdash t.j : A_j}$$

$$\frac{\Gamma \vdash t : \prod_{i \in J} A_i \qquad I \subseteq J}{\Gamma \vdash t|_{I \subseteq J} : \prod_{i \in I} A_i} \qquad \frac{\Gamma \vdash t : \prod_{i \in I} A_i \qquad J = \{i_1, \ldots, i_n\} \qquad I \# J \qquad \forall j \in J. \, \Gamma \vdash v_{\mho j} : A_j}{\Gamma \vdash \{t ; j_1 = v_{\mho 1}, \ldots, j_n = v_{\mho n}\} : \prod_{i \in I \uplus J} A_i}$$

Fig. 20. Core Language Typing

(3) We denote by ePreDom the category of error predomains and continuous functions, with the obvious identities and composition.

We can show that our category of predomains is very rich by noticing that it is monadic over a category of "double preorders", just as dcpos are monadic over preorders by the ideal completion monad.

$$\frac{}{\mathrm{E}[\mathrm{unroll}_? \; \mathrm{roll}_? \; \mathrm{v}] \longmapsto \mathrm{E}[\mathrm{v}]} \qquad \frac{}{\mathrm{E}[(\mathrm{rec}\,\mathrm{x}\,\mathrm{y}\,.\,\mathrm{t})\;\mathrm{v}] \longmapsto \mathrm{E}[\mathrm{t}[\mathrm{rec}\,\mathrm{x}\,\mathrm{y}\,.\,\mathrm{t}/\mathrm{x}][\mathrm{v}/\mathrm{y}]]}$$

$$\frac{}{\mathrm{E}[\mathrm{case}\,(\mathrm{inj}_\mathrm{i}\;\mathrm{v})\;\mathrm{of}\;\mathrm{x}_1.\,\mathrm{t}_1 \mid \mathrm{x}_2.\,\mathrm{t}_2] \longmapsto \mathrm{E}[\mathrm{t}_i[\mathrm{v}/\mathrm{x}_i]]} \qquad \frac{}{\mathrm{E}[\pi_{\mathrm{i}_k}\,\langle \mathrm{v}_{\mathrm{i}_1},\ldots,\mathrm{v}_{\mathrm{i}_n}\rangle_\mathrm{I}] \longmapsto \mathrm{E}[\mathrm{v}_{\mathrm{i}_k}]}$$

$$\frac{\mathrm{i} \in \{\mathrm{i}_1,\ldots,\mathrm{i}_n\}}{\mathrm{E}[\{\mathrm{v}_{\mathrm{i}_1},\ldots,\mathrm{v}_{\mathrm{i}_n}\}_\mathrm{I}.\mathrm{i}] \longmapsto \mathrm{E}[\mathrm{v}_\mathrm{i}]} \qquad \frac{\mathrm{i} \notin \{\mathrm{i}_1,\ldots,\mathrm{i}_n\}}{\mathrm{E}[\{\mathrm{v}_{\mathrm{i}_1},\ldots,\mathrm{v}_{\mathrm{i}_n}\}_\mathrm{I}.\mathrm{i}] \longmapsto \mathrm{E}[\mho]} \qquad \frac{}{\mathrm{E}[(\lambda \mathrm{x}.\,\mathrm{t})\;\mathrm{v}] \longmapsto \mathrm{E}[\mathrm{t}[\mathrm{v}/\mathrm{x}]]}$$

$$\frac{}{\mathrm{E}[\mathrm{let}\;\mathrm{x} = \mathrm{v}\;\mathrm{in}\;\mathrm{t}] \longmapsto \mathrm{E}[\mathrm{t}[\mathrm{v}/\mathrm{x}]]} \qquad \frac{}{\mathrm{E}[\mho] \longmapsto \mho}$$

Fig. 21. Core Language Operational Semantics

Definition A.2 (Error Ideal). For a double preorder $P = (|P|, \le_P, \sqsubseteq_P)$, an *error ideal* of P is a subset $D \subset |P|$ that is

(1) Directed in $\le_P$: If $D' \subset D$ is a finite subset, then D contains an upper bound $d \in D$, $\forall d' \in D'.d' \le_P d$ (with $D' = \emptyset$ this shows D is inhabited).
(2) Downward closed in $\le_P$: if $d \in D, d' \le_P d$ then $d' \in D$

Then $\mathrm{EIdl}(P)$ is a predomain with

(1) $D \le_{\mathrm{EIdl}(P)} E$ when $D \subseteq E$
(2) $D \sqsubseteq_{\mathrm{EIdl}(P)} E$ when for any $d \in D, e \in E$, there exists an $d' \in D$ $e' \in E$ with $d \le_X d', e \le_X e'$ such that $d' \sqsubseteq_X e'$.

That is, no matter how far we have "advanced" in the directed sets D, E there are later elements with $d' \sqsubseteq_X e'$.

We can justify that the definition of $\sqsubseteq_{\mathrm{EIdl}(P)}$ is the right (best) one by observing that $D \sqsubseteq_{\mathrm{EIdl}(P)} E$ if and only if there is a directed set I and nets $\delta : I \to D, \epsilon : I \to E$ whose downward closures are D, E respectively where for every $i \in I$, $\delta(i) \sqsubseteq_P \epsilon(i)$. If D, E are represented ω-chains, this means $D \sqsubseteq_{\mathrm{EIdl}(P)} E$ if and only if we can construct subsequences d_i, e_i that are $\sqsubseteq_P$ in lock-step. The indexing set is the set of all finite subsets of D, E, then the nets can be constructed using the axiom of choice.

Note that the downward-closure condition is primarily to make $\le_{\mathrm{EIdl}(P)}$ a poset, so by not requiring a similar condition, $\sqsubseteq_{\mathrm{EIdl}(P)}$ is in fact a preorder. We cannot add in a similar saturation condition for $\sqsubseteq_{\mathrm{EIdl}(P)}$ because it would necessitate new elements due to the directedness and the independence of $\le_P, \sqsubseteq_P$. Instead, to achieve a poset, we should quotient error ideals by $\sqsubseteq_{\mathrm{EIdl}(P)}$.

Theorem A.3 (ePreDom is Monadic over Double Preorders). (1) *$\mathrm{EIdl}(\cdot)$ extends to a monad on the category of double preorders and doubly monotone functions.*
(2) *The category ePreDom is equivalent to the category of algebras of the error ideal monad.*

Proof. (1) First we show EIdl is a functor. Let $f : P \to Q$ be a doubly monotone function. Define $\mathrm{EIdl}(f)(D) = \downarrow\{f(d) | d \in D\}$. Then f clearly is monotone and continuous wrt $\le_{\mathrm{EIdl}(P)}$. We need to show f is monotone wrt $\sqsubseteq_{\mathrm{EIdl}(P)}$.

Let $D \sqsubseteq_{\mathrm{EIdl}(P)} E$ and $a \in \mathrm{EIdl}(f)(D), b \in \mathrm{EIdl}(f)(E)$. Then there exist $d \in D, e \in E$ with $fd \le_Q a, fe \le_Q b$ and $fd \in \mathrm{EIdl}(f)(D), \mathrm{EIdl}(f)(E)$. Since $D \sqsubseteq_{\mathrm{EIdl}(P)} E$ there exist further $d' \in D, e' \in E$ with $d' \sqsubseteq_P e'$ and $d \le_P d', e \le_P e'$. Then by monotonicity of f

$$f(d') \sqsubseteq_Q f(e')$$

$$
\begin{aligned}
?^{+} &\overset{\text{def}}{=} ? \\
\mathbb{N}^{+} &\overset{\text{def}}{=} \mathbb{N} \\
0^{+} &\overset{\text{def}}{=} \mathbf{0} \\
(\mathsf{A}_1 + \mathsf{A}_2)^{+} &\overset{\text{def}}{=} \mathsf{A}_1{}^{+} + \mathsf{A}_2{}^{+} \\
1^{+} &\overset{\text{def}}{=} \mathbf{1} \\
(\mathsf{A}_1 \times \mathsf{A}_2)^{+} &\overset{\text{def}}{=} \mathsf{A}_1{}^{+} \times \mathsf{A}_2{}^{+} \\
(\mathsf{A}_1 \to \mathsf{A}_2)^{+} &\overset{\text{def}}{=} \mathsf{A}_1{}^{+} \to \mathsf{A}_2{}^{+} \\
(\{\eta_1 : \mathsf{A}_{\eta_1}, \ldots, \eta_n : \mathsf{A}_{\eta_n}\})^{+} &\overset{\text{def}}{=} \bigtimes_{\eta \in \{\eta_1, \ldots, \eta_n\}} \mathsf{A}_\eta{}^{+} \\
(\{\eta_1 : \mathsf{A}_{\eta_1}, \ldots, \eta_n : \mathsf{A}_{\eta_n}, ?\})^{+} &\overset{\text{def}}{=} \bigtimes_{\eta \in \{\eta_1, \ldots, \eta_n\}} \mathsf{A}_\eta{}^{+} \times \prod_{- \in \mathfrak{F} \setminus \{\eta_1, \ldots, \eta_n\}} ? \\
\mathsf{A}^{\mathbf{e},\mathbf{p}} &\overset{\text{def}}{=} \mathsf{Imp}(\mathsf{A})^{\mathbf{e},\mathbf{p}} \\
(\mathsf{A} \implies \mathsf{B})^{+} &\overset{\text{def}}{=} \mathsf{B}^{\mathbf{p}}[\mathsf{A}^{\mathbf{e}}] \\
\\
\mathsf{id_A}^{\mathbf{e},\mathbf{p}} &\overset{\text{def}}{=} [\cdot] \\
(\mathsf{d};\mathsf{d}')^{\mathbf{e}} &\overset{\text{def}}{=} \mathsf{d}'^{\mathbf{e}}[\mathsf{d}^{\mathbf{e}}] \\
(\mathsf{d};\mathsf{d}')^{\mathbf{p}} &\overset{\text{def}}{=} \mathsf{d}^{\mathbf{p}}[\mathsf{d}'^{\mathbf{p}}] \\
\mathsf{Tag_G}^{\mathbf{e}} &\overset{\text{def}}{=} \mathbf{inj}_{\mathsf{G}^{+}}\, [\cdot] \\
\mathsf{Tag_G}^{\mathbf{p}} &\overset{\text{def}}{=} \mathbf{case}\, [\cdot]\, \mathbf{of}\, \mathbf{inj}_{\mathsf{G}^{+}}\, \mathsf{x}.\, \mathsf{x} \mid \mathbf{else}.\, \mho \\
(\mathsf{d} + \mathsf{d}')^{\mathbf{e},\mathbf{p}} &\overset{\text{def}}{=} \mathsf{d}^{\mathbf{e},\mathbf{p}} + \mathsf{d}'^{\mathbf{e},\mathbf{p}} \\
(\mathsf{d} \times \mathsf{d}')^{\mathbf{e},\mathbf{p}} &\overset{\text{def}}{=} \mathsf{d}^{\mathbf{e},\mathbf{p}} \times \mathsf{d}'^{\mathbf{e},\mathbf{p}} \\
(\mathsf{d} \to \mathsf{d}')^{\mathbf{e},\mathbf{p}} &\overset{\text{def}}{=} \mathsf{d}^{\mathbf{p},\mathbf{e}} \to \mathsf{d}'^{\mathbf{e},\mathbf{p}} \\
\{\eta_1 : \mathsf{d}_1, \ldots, \eta_n : \mathsf{d}_n\}^{\mathbf{e},\mathbf{p}} &\overset{\text{def}}{=} \bigtimes_{\mathrm{i} \in 1, \ldots, n} \mathsf{d}_i{}^{\mathbf{e},\mathbf{p}} \\
\{\eta_1 : \mathsf{d}_1, \ldots, \eta_n : \mathsf{d}_n, ?\}^{\mathbf{e},\mathbf{p}} &\overset{\text{def}}{=} (\{\eta_1 : \mathsf{d}_1, \ldots, \eta_n : \mathsf{d}_n\}^{\mathbf{e},\mathbf{p}}) \times [\cdot] \\
\mathsf{PrecGrad}\langle\rho\rangle^{\mathbf{e}} &\overset{\text{def}}{=} \langle [\cdot], \{\}_{\mathfrak{F} - \rho} \rangle \\
\mathsf{PrecGrad}\langle\rho\rangle^{\mathbf{p}} &\overset{\text{def}}{=} \pi_1\, [\cdot] \\
\mathsf{Width}_?\langle\rho;\eta_1, \ldots, \eta_n\rangle^{\mathbf{e}} &\overset{\text{def}}{=} \mathbf{let}\, \langle \mathbf{x}_p, \mathbf{x}_g \rangle = [\cdot]\, \mathbf{in} \\
&\phantom{\overset{\text{def}}{=}} \mathbf{let}\, \mathbf{x}_{\eta_1} = \pi_{\eta_1}\, \mathbf{x}_p \cdots \mathbf{in} \\
&\phantom{\overset{\text{def}}{=}} \left\langle \begin{array}{l} \langle \eta_1' = \pi_{\eta_1'}\, \mathbf{x}_p, \ldots \rangle_\rho, \\ \qquad \{\mathbf{x}_g\, ; \eta_1 = \mathbf{x}_{\eta_1}, \ldots\} \end{array} \right\rangle \\
\mathsf{Width}_?\langle\rho;\eta_1, \ldots, \eta_n\rangle^{\mathbf{p}} &\overset{\text{def}}{=} \mathbf{let}\, \langle \mathbf{x}_p, \mathbf{x}_g \rangle = [\cdot]\, \mathbf{in} \\
&\phantom{\overset{\text{def}}{=}} \left\langle \begin{array}{l} \left\langle \begin{array}{l} \eta_1' = \pi_{\eta_1'}\, \mathbf{x}_p, \ldots, \\ \eta_1 = \mathbf{x}_g.\eta_1, \ldots \end{array} \right\rangle_{\rho, \eta_1, \ldots}, \\ \qquad \mathbf{x}_g|_{\mathfrak{F} - \rho, \eta_1, \ldots \subseteq \mathfrak{F} - \rho} \end{array} \right\rangle
\end{aligned}
$$

Fig. 22. Source to Core Type and Type Precision Elaborator

and by transitivity, these elements are bigger by $\leq_Q$ than a, b respectively.

Next, the unit of the monad is the principal ideal (downward-closure) function $\downarrow x = \{y \in P | y \leq_P x\}$, which is clearly monotone in $\leq_P$ and is monotone in $\sqsubseteq_P$ since we can always choose the top elements of the principal ideal in the proof.

Finally the join of the monad is just the union of the ideal of ideals. To show this is monotone in $\sqsubseteq_{\mathrm{EIdl}(\mathrm{EIdl}P)}$, let $\mathcal{D} \sqsubseteq_{\mathrm{EIdl}(\mathrm{EIdl}P)} \mathcal{E}$. Then for any $d \in \bigcup \mathcal{D}, e \in \bigcup \mathcal{E}$ there are

$$
\begin{array}{rcl}
(\mathsf{x})^{+} & \overset{\text{def}}{=} & \mathbf{x} \\
(\Uparrow^{?}_{A}\ (\mathsf{t}))^{+} & \overset{\text{def}}{=} & A^{\mathbf{e}}[(\mathsf{t})^{+}] \\
(\Downarrow^{?}_{A}\ (\mathsf{t}))^{+} & \overset{\text{def}}{=} & A^{\mathbf{p}}[(\mathsf{t})^{+}] \\
(\mathsf{roll}_{?}\ \mathsf{t})^{+} & \overset{\text{def}}{=} & \mathbf{roll}_{?}\ (\mathsf{t})^{+} \\
(\mathsf{unroll}_{?}\ (\mathsf{t}\ @\ A))^{+} & \overset{\text{def}}{=} & \mathbf{unroll}_{?}\ A^{\mathbf{e}}[(\mathsf{t})^{+}] \\
(\mathsf{rec}\ \mathsf{x}\ \mathsf{y}\,.\,\mathsf{t})^{+} & \overset{\text{def}}{=} & \mathbf{rec}\ \mathbf{x}\ \mathbf{y}\,.\,(\mathsf{t})^{+} \\
(\mathsf{n})^{+} & \overset{\text{def}}{=} & \mathbf{n} \\
(\mathsf{add1}\ (\mathsf{t}\ @\ A))^{+} & \overset{\text{def}}{=} & \mathbf{add1}\ ((A \implies \mathbb{N})^{+}[(\mathsf{t})^{+}]) \\
(\mathsf{sub1}\ (\mathsf{t}\ @\ A))^{+} & \overset{\text{def}}{=} & \mathbf{sub1}\ ((A \implies \mathbb{N})^{+}[(\mathsf{t})^{+}]) \\
(\mathsf{case}\ (\mathsf{t}\ @\ A \Rightarrow 0)\ \mathsf{of}\ ())^{+} & \overset{\text{def}}{=} & \mathbf{case}\ (A \implies 0)^{+}[(\mathsf{t})^{+}]\ \mathbf{of}\ () \\
(\mathsf{case}\ (\mathsf{t}\ @\ A \Rightarrow A_1 + A_2)\ \mathsf{of}\ \mathsf{x}_1.\,\mathsf{t}_1 \mid \mathsf{x}_2.\,\mathsf{t}_2)^{+} & \overset{\text{def}}{=} & \mathbf{case}\ (A \implies (A_1 + A_2))^{+}[(\mathsf{t})^{+}]\ \mathbf{of}\ \mathbf{x}_1.\,(\mathsf{t}_1)^{+} \mid \mathbf{x}_2.\,(\mathsf{t}_2)^{+} \\
(\mathsf{inj}_{\mathsf{i}}\ \mathsf{t})^{+} & \overset{\text{def}}{=} & \mathbf{inj}_{\mathbf{i}}\ (\mathsf{t})^{+} \\
(\langle\rangle)^{+} & \overset{\text{def}}{=} & \langle\rangle \\
(\langle \mathsf{t}_1, \mathsf{t}_2 \rangle)^{+} & \overset{\text{def}}{=} & \langle (\mathsf{t}_1)^{+}, (\mathsf{t}_2)^{+} \rangle \\
(\pi_1\ (\mathsf{t}\ @\ A \Rightarrow A_1 \times ?))^{+} & \overset{\text{def}}{=} & \pi_1\ ((A \implies (A_1 \times ?))^{+}[(\mathsf{t})^{+}]) \\
(\pi_2\ (\mathsf{t}\ @\ A \Rightarrow ? \times A_2))^{+} & \overset{\text{def}}{=} & \pi_2\ ((A \implies (? \times A_2))^{+}[(\mathsf{t})^{+}]) \\
(\lambda \mathsf{x}.\,\mathsf{t})^{+} & \overset{\text{def}}{=} & \lambda \mathbf{x}.\,(\mathsf{t})^{+} \\
((\mathsf{t}\ @\ A_t \Rightarrow A_u \to B)\ \mathsf{u})^{+} & \overset{\text{def}}{=} & ((A_t \implies (A_u \to B))^{+}[(\mathsf{t})^{+}])\ \mathsf{u}
\end{array}
$$

Fig. 23. Source to Core Term Elaborator

ideals with $d \in D \in \mathcal{D}, e \in E \in \mathcal{E}$ since $\mathcal{D} \sqsubseteq_{\mathrm{EIdl}(P)} \mathcal{E}$, there are bigger ideals D', E' with $D' \sqsubseteq_{\mathrm{EIdl}(P)} E'$ so there are $d \leq_P d' \in D', e \leq_P e' \in E'$ with $d' \sqsubseteq_P e'$.

(2) Given any predomain X, the directed join function is easily seen to be an algebra $\bigvee : \mathrm{EIdl}(X) \to X$, and an algebra homomorphism of such algebras is exactly a continuous function of predomains.

To complete the proof, we show that *any* algebra for $\mathrm{EIdl}(\cdot)$ must be the directed join, the proof is an instance of [18]. Let $\alpha : \mathrm{EIdl}(P) \to P$ be an algebra of $\mathrm{EIdl}(\cdot)$. The directed join operation has the universal property of being a left adjoint to the principal ideal map. Thus to show $\alpha = \bigvee$ it is sufficient to show that it is a left adjoint of $\downarrow : P \to \mathrm{EIdl}(P)$, then monotonicity of α wrt $\sqsubseteq_{\mathrm{EIdl}(P)}$ means the error ordering on P is compatible with directed joins as in the definition of predomains.

In more pedestrian terms, the directed join is characterized by the property that for any ideal $D \bigvee D \leq_P x$ if and only if $y \leq_P x$ for every $y \in D$, which is the same as saying $D \subseteq \downarrow x$. To show that a retract of the principal ideal map $\alpha : \mathrm{EIdl}(P) \to P$ has this property, it is sufficient to show that $D \subseteq \downarrow(\alpha(D))$. For the reverse, if $D \subseteq \downarrow x$, then $\alpha(D) \leq_P \alpha(\downarrow x) = x$. For the forward direction, assume $\alpha(D) \leq_P x$, then $D \subseteq \downarrow \alpha(D) \subseteq \downarrow(x)$

To prove that $D \subseteq \downarrow(\alpha(D))$, first observe that

$$\mathrm{EIdl}(\downarrow D) = \{E \in \mathrm{EIdl}(P) | \exists d \in D, E \subseteq \downarrow(d)\} \subseteq \downarrow(D) = \{E \in \mathrm{EIdl}(P) | E \subseteq D\}$$

since $\downarrow(d) \subseteq D$ for any $d \in D$. Applying $\mathrm{EIdl}(\alpha)$ to both sides we get $\mathrm{EIdl}(\alpha)(\mathrm{EIdl}(\downarrow)(D)) = \mathrm{EIdl}(\alpha \circ \downarrow) = \mathrm{EIdl}(\mathrm{id}_P) = \mathrm{id}_{\mathrm{EIdl}(P)}$ by functoriality and that α is an algebra.

On the other side $\mathrm{EIdl}(\alpha)(\downarrow_{\mathrm{EIdl}(P)}(D)) = \downarrow_P(\alpha(D))$ by naturality of $\downarrow$. So since $\mathrm{EIdl}(\downarrow)(D) \subseteq \downarrow(D)$, by monotonicity of α, we conclude

$$D = \mathrm{EIdl}(\alpha)(\mathrm{EIdl}(\downarrow)(D)) \subseteq \mathrm{EIdl}(\alpha)(\downarrow(D)) = \downarrow(\alpha(D))$$

□

Corollary A.4 (ePreDom has all countable limits). *Furthermore, the underlying set of any limit is the limit of the diagram of underlying sets.*

Proof. Since ePreDom is monadic over double preorders, the forgetful functor creates all limits from double preorders (). Since the category of double preorders has all countable limits (in fact arbitrary limits), ePreDom has them as well.

Furthermore, since the underlying set functor from double preorders to sets has a left adjoint (the discrete double poset), the underlying set of the limit is the limit of the underlying sets. This means we can construct all limits by first constructing them on the underlying sets and then giving them the least ordering that makes all projections monotone. □

Corollary A.5 (ePreDom has finite coproducts). *Furthermore, the underlying set of any finite coproduct is the coproduct of the diagram of underlying sets.*

Proof. Since ePreDom is monadic over double preorders, the forgetful functor creates all colimits that commute with the error ideal monad.

$\mathrm{EIdl}(\emptyset) = \emptyset$ since any ideal must be inhabited. $\mathrm{EIdl}(P + Q) \equiv \mathrm{EIdl}(P) + \mathrm{EIdl}(Q)$ since $\mathrm{inl}(p), \mathrm{inr}(q)$ have no upper bound.

Furthermore, the underlying set of a finite coproduct of double preorders is the disjoint union of sets or empty set. □

Next, to interpret our effects divergence and errors we define appropriate notions of *domain*. Here things become more complicated than usual domain theory, since we have 2 effects: divergence and errors.

To this end, we define 2 notions of "domain": one for divergence, and one for both divergence and errors.

Ordinary "domain theory", i.e., for the purposes of taking fixed points of functions and functors, will only involve the divergence bottom Ω and the error ordering $\sqsubseteq_{\mathrm{err}}$ is just "along for the ride".

However, since we do not have an effect system, arbitrary terms admit the possibility of error or divergence, so for the semantics we will primarily be interested in domains that have least elements for both orderings, so we call these "domains".

Definition A.6 (Divergence Domains, Domains).

(1) A divergence domain is a tuple $(|Y|, \leq_Y, \sqsubseteq_Y, \Omega_Y)$ such that $(|Y|, \leq_Y, \sqsubseteq_Y)$ is a predomain and $\Omega_Y \in |Y|$ is the least element by $\leq_Y$. We denote by $U_\Omega(Y)$ the underlying predomain.
(2) Given Y, Y' divergence domains, a strict function $f : Y \to_s Y'$ is a function $|f| : |Y| \to |Y'|$ of underlying sets so that f is a continuous function of the underlying predomains and $f(\Omega_Y) = f(\Omega_{Y'})$. We denote the underlying continuous function of predomains as $U_\Omega(f) : U_\Omega(Y) \to U_\Omega(Y')$.
(3) We denote by divDom_s the category of divergence domains and strict maps. Then U_Ω is a functor $U_\Omega : \mathrm{divDom}_s \to \mathrm{ePreDom}$.
(4) A domain is a tuple $(|Y|, \leq_Y, \sqsubseteq_Y, \Omega_Y, \mho_Y)$ such that $(|Y|, \leq_Y, \sqsubseteq_Y)$ is a predomain and $\Omega_Y \in |Y|$ is the least element by $\leq_Y$ and $\mho_Y \in |Y|$ is the least element by $\mho_Y$. We denote by $U_\mho(Y)$ the underlying divergence domain.

(5) Given Y, Y' domains, a doubly strict (continuous) function $f : Y \to_{ss} Y'$ is a function $|f| : |Y| \to |Y'|$ of underlying sets so that f is a continuous function of the underlying predomains, $f(\Omega_Y) = f(\Omega_{Y'})$ and $f(\mho_Y) = \mho_{Y'}$. We denote the underlying strict function of divergence domains as $U_\mho(f) : U_\mho(Y) \to U_\mho(Y')$.

(6) We denote by divDom_s the category of divergence domains and strict maps. Then U_Ω is a functor $U_\Omega : \mathrm{divDom}_s \to \mathrm{ePreDom}$.

As expected, these $U_\Omega, U_\mho$ have right adjoints that produce the expected "lifting" monads on PreDom.

Definition A.7 (Free Divergence Domains, Domains). (1) We denote by $F_\Omega : \mathrm{PreDom} \to \mathrm{divDom}_s$ the free divergence domain from a predomain constructed as:

$$F_\Omega(|X|, \leq_X, \sqsubseteq_X) = (|X| + \{\Omega\}, \leq_{F_\Omega(X)}, \sqsubseteq_{F_\Omega(X)}, \mathrm{inr}(\Omega))$$

with orderings

$$\begin{aligned} p \leq_{F_\Omega(X)} q \quad &\text{iff} \quad p = \mathrm{inr}(\Omega) \vee (p = \mathrm{inl}(x), q = \mathrm{inl}(y), x \leq_X y) \\ p \sqsubseteq_{F_\Omega(X)} q \quad &\text{iff} \quad p = q = \mathrm{inr}(\Omega) \vee (p = \mathrm{inl}(x), q = \mathrm{inl}(y), x \sqsubseteq_X y) \end{aligned}$$

(2) F_Ω extends to a functor that is left adjoint to U_Ω

(3) We denote by $F_\mho : \mathrm{divDom}_s \to \mathrm{Dom}_{ss}$ the free domain from a divergence domain constructed as:

$$F_\mho(|X|, \leq_X, \sqsubseteq_X, \Omega_X) = (|X| + \{\mho\}, \leq_{F_\mho(X)}, \sqsubseteq_{F_\mho(X)}, \mathrm{inl}(\Omega_X), \mathrm{inr}(\mho))$$

with orderings

$$\begin{aligned} p \leq_{F_\Omega(X)} q \quad &\text{iff} \quad p = q = \mathrm{inr}(\mho) \vee (p = \mathrm{inl}(x), q = \mathrm{inl}(y), x \leq_X y) \\ p \sqsubseteq_{F_\Omega(X)} q \quad &\text{iff} \quad p = \mathrm{inr}(\mho) \vee (p = \mathrm{inl}(x), q = \mathrm{inl}(y), x \sqsubseteq_X y) \end{aligned}$$

(4) $F_\mho$ extends to a functor that is left adjoint to $U_\mho$

Definition A.8 (Error Domains, Strict Continuous Functions). (1) An error domain is a tuple $(|Y|, \leq_Y, \sqsubseteq_Y, \Omega_Y, \mho_Y)$ where $(|Y|, \leq_Y, \sqsubseteq_Y)$ is an error predomain, $\Omega_Y \in |Y|$ is the least element of the divergence predomain and $\mho_Y \in |Y|$ is the least element of the error preorder. We denote by $U(|Y|, \leq_Y, \sqsubseteq_Y, \Omega_Y, \mho_Y) = (|Y|, \leq_Y, \sqsubseteq_Y)$ the underlying error predomain. This makes $(|Y|, \leq_Y, \Omega_Y)$ a domain and $(|Y|, \sqsubseteq_Y, \mho_Y)$ a preorder with least element.

(2) A (doubly) strict continuous function $f : Y \to_s Y'$ where Y, Y' are error domains is a continuous function of error predomains $f : Y \to Y'$ so that $f : (|Y|, \leq_Y, \Omega_Y) \to (|Y'|, \leq_{Y'}, \Omega_{Y'})$ is continuous as a morphism of error predomains, $f(\Omega_Y) = \Omega_{Y'}$ and and $f(\mho_Y) = \mho_{Y'}$. This makes f a strict continuous function of the underlying divergence domain and a strict monotone function of the underlying error preorder. We denote by $U(f : Y \to_s Y') = f : U(Y) \to U(Y')$ the underlying continuous function of error predomains.

(3) We denote by eDom_s the category of error domains with strict continuous functions, with the obvious identity and composition.

(4) With the above definition U is a functor $U : \mathrm{eDom}_s \to \mathrm{ePreDom}$, that is it preserves identity and composition.

A.2.1 Recursive Type. Next we calculate solutions to recursive domain equations. This will be used to construct the denotation of the dynamic type ? and also the logical relation for adequacy. The development mostly follows [20], but adjusted to maintain a more thorough separation of predomains and domains.

Definition A.9 (Divergence Kleisli Category). We denote by PreDom_Ω the klesli category of the adjunction $F_\Omega \dashv U_\Omega$. That is the objects are predomains and the morphisms $X_1 \to_\Omega X_2$ are continuous functions $X_1 \to U_\Omega F_\Omega X_2$ or equivalently strict continuous functions $F_\Omega X_1 \to_{ss} F_\Omega X_2$.

The identity can be viewed as the unit of the monad $\lfloor\cdot\rfloor : X \to U_\Omega F_\Omega X$ or the identity strict function $\mathrm{id} : F_\Omega X \to F_\Omega X$.

Composition can be viewed as either the Kleisli composition $f \circ g = \mu \circ U_\Omega F_\Omega(g) \circ f)$ where μ is the multiplication of the monad or simply as composition of strict functions $f \circ g$.

As much as possible, we will keep abstract the precise domain constructions, and only work up to the interface provided by the notion of "minimal invariant". To do this we need some basic notions from enriched-category theory. Specifically, we note that PreDom_Ω is a *CPO*-enriched category.

Definition A.10 (CPO-Categorical properties of PreDom_Ω). (1) Each hom set $X_1 \to_\Omega X_2$ is a CPO with least element $\lambda x.\Omega$ using the divergence ordering.
(2) Composition $\circ : (X_1 \to_\Omega X_2) \times (X_2 \to_\Omega X_3) \to (X_1 \to X_3)$ is continuous and strict in the post-composing argument, i.e., $\Omega \circ f = \Omega$.

Proof. Immediate by the fact that all above constructions factor through a forgetful functor to the analogous category for CPOs that has all of these properties. □

Then to solve recursive domain equations we need two simple ingredients about PreDom_Ω.

Definition A.11 (PreDom_Ω Limits). (1) 0 is a terminal (in fact zero) object in PreDom_Ω.
(2) PreDom_Ω has ω^{op} limits

Proof. (1) Obvious.
(2) Let $\{X_i, f_i\}_{i\in\omega}$ be an ω^{op} diagram in PreDom_Ω, that is, $f_i : X_{i+1} \to_\Omega X_i$. Let L be the limit of the corresponding diagram $\{U_\Omega F_\Omega(X_i), g_i\}$ where $g_i : U_\Omega F_\Omega X_{i+1} \to U_\Omega F_\Omega X_i$ is given as $g_i = U_\Omega(f_i)$. Denote the projections as $\pi_i : L \to X_i$. By an argument given in [28], L has a divergence-least element and the π_i preserve it, so L, π_i are a limit of the corresponding diagram in divDom_s since U_Ω is fully faithful.

Since there is a fully faithful forgetful functor from PreDom_Ω to divDom_s, it is sufficient to show that L is isomorphic in divDom_s to $F_\Omega(L')$ for some predomain L'. We define $|L'| = \{x \in |L| | x \neq \Omega_L\}$ with divergence and error orderings given by the orderings on L'. Then the following f, f^{-1} form a strict isomorphism between L and $F_\Omega(L)$ in divDom_s (using the law of excluded middle):

$$f(x) = \begin{cases} \Omega_{F_\Omega L} & \text{if } x = \Omega_{L'} \\ x & \text{if } x \neq \Omega_{L'} \end{cases}$$
$$f^{-1}(x) = \begin{cases} \Omega_{L} & \text{if } x = \Omega_{L'} \\ x & \text{if } x \neq \Omega_{L'} \end{cases}$$

□

Finally we define minimal invariants and show their existence for functors of interest.

Definition A.12 (Minimal Invariant). Let $C : \mathrm{PreDom}_\Omega^{op} \times \mathrm{PreDom}_\Omega \to \mathrm{PreDom}_\Omega$ be a locally continuous functor. A minimal invariant for C is a predomain D and an isomorphism (in PreDom) $i : C(D, D) \cong D$ such that id_D is the $\leq_D$-least (in fact, unique) fixed point of $\delta : (D \to_\Omega D) \to (D \to_\Omega D)$ defined by

$$\delta(e) = i \circ C(e, e) \circ i^{-1}$$

More explicitly $\lim_{n\to\omega} \delta^n(\bot) = \mathrm{id}_D$.

Theorem A.13 (Minimal Invariants Always Exist). *Every locally continuous functor $C : PreDom_{\Omega}^{op} \times PreDom_{\Omega} \to PreDom_{\Omega}$ has a unique minimal invariant.*

Proof. PreDom_{Ω} meets the criteria for the theorems as stated in [12, 20, 28]. We summarize them as follows (based on amadio-curien):

(1) PreDom_{Ω} is CPPO-enriched, with composition strict in the later morphism.
(2) PreDom_{Ω} has a terminal object and limits of ω^{op} chains.

Then we can use the classic construction and this produces a minimal invariant.

Their uniqueness is a consequence of their characterization as an initial algebra/final coalgebra due to Freyd. □

Theorem A.14 (Locally Continuous Functors). *The following are locally continuous functors*

(1) *empty product* $1 : 1 \to PreDom_{\Omega}$
(2) *binary product* $X_1 \times X_2 : (PreDom_{\Omega}^{op})^0 \times PreDom_{\Omega}^2 \to PreDom_{\Omega}$
(3) *empty sum* $0 : 1 \to PreDom_{\Omega}$
(4) *binary sums* $X_1 + X_2 : (PreDom_{\Omega}^{op})^0 \times PreDom_{\Omega}^2 \to PreDom_{\Omega}$
(5) *erroring, diverging functions* $U_{\Omega}(X_1 \to U_{\mho}F_{\mho}F_{\Omega}(X_2)) : (PreDom_{\Omega}^{op})^1 \times PreDom_{\Omega}^1 \to PreDom_{\Omega}$

Proof. Most are obvious, we consider products and functions.

(1) For the binary product let $f : X_1 \to_{\Omega} Y_1, g : X_2 \to_{\Omega} Y_2$, define $(f \times_k g) = \beta \circ (f \times g)$ where $f \times g : X_1 \times X_2 \to (U_{\Omega}F_{\Omega}Y_1 \times U_{\Omega}F_{\Omega}Y_2)$ is given by functoriality of $\times$ on PreDom and $\beta : (U_{\Omega}F_{\Omega}Y_1 \times U_{\Omega}F_{\Omega}Y_2) \to U_{\Omega}F_{\Omega}(Y_1 \times Y_2)$ is the natural transformation exhibiting $U_{\Omega}F_{\Omega}$ as a commutative monad, defined by

$$\beta(\Omega, -) = \Omega$$

$$\beta(-, \Omega) = \Omega$$

$$\beta(\lfloor x \rfloor, \lfloor y \rfloor) = (x, y)$$

Local continuity follows from the fact that $\times$ is locally continuous on PreDom.

(2) For functions, let $f : Y_1 \to_{\Omega} X_1, g : X_2 \to_{\Omega} Y_2$, define $(f \to U_{\mho}F_{\mho}F_{\Omega}g)(h) = U_{\mho}F_{\mho}(g) \circ h \circ f$, considering f, g as strict continuous functions. Functoriality is obvious, local continuity follows from local continuity for $U_{\mho}F_{\mho}$.

□

Definition A.15 (Denotation of the dynamic type). We define $Dyn : (\mathrm{PreDom}_{\Omega}^{op})^1 \times \mathrm{PreDom}_{\Omega}^1 \to \mathrm{PreDom}_{\Omega}$ to be the locally continuous functor given by

$$Dyn(D^-, D^+) = 1 + (D^+ + D^+) + (D^+ \times D^+) + (D^- \to U_{\mho}F_{\mho}F_{\Omega}D^+)$$

Then we define $?, i : Dyn(?, ?) \cong ?$ to be a minimal invariant.

A.2.2 *Denotational Semantics.*

We now present the denotational semantics for the core language. In the next subsection we prove soundness and adequacy of the semantics.

We interpret terms in the Kleisli category of the adjunction $F_{\Omega,\mho} \dashv U_{\Omega,\mho}$ which we describe now:

Definition A.16 (Category of Diverging-Erroring Functions). The category of predomains and diverging, erroring functions $\mathrm{PreDom}_{\Omega,\mho}$ is defined as the Kleisli category of the adjunction $F_{\Omega,\mho} \dashv U_{\Omega,\mho}$ with objects predomains and arrows $X \to_{\Omega,\mho} Y$ equivalently

(1) Continuous functions $X \to U_{\Omega,\mho}F_{\Omega,\mho}Y$
(2) Doubly strict continuous functions $F_{\Omega,\mho}X \to_{ss} F_{\Omega,\mho}Y$.

A type $\mathbf{A}$ denotes a pre-domain $[\![\mathbf{A}]\!]$. The sums and products are defined by their universal property. The natural numbers are the discrete pre-domain of natural numbers. The function type uses the "error-divergence lift". The dynamic type is defined as a limit of a certain sequence of embedding-projection pairs.

The most peculiar type in our core language is the partial product type $\{\mathbf{I}\}_{\mathbf{i}}\mathbf{A}_i$. Semantically, it is just the type of partial functions from $\mathbf{i} \in \mathbf{I}$ to $\mathbf{A}_i$, where partiality is modelled by $\mho$: $\prod_{\mathbf{i}\in\mathbf{I}} [\![\mathbf{A}_i]\!]_{\mho}$. We refine this slightly by considering only those products with finite support, where the support is defined as

$$\mathrm{supp}(p \in \prod_{i\in I} X_{i\mho}) = \{j \in I | \exists x.p(j) = \lfloor x \rfloor\}$$

We observe that this finite support property would not produce a predomain if we were using Ω to represent partiality, since it would not have limits of increasingly more defined functions, but there is no difficulty with $\mho$ as partiality since predomains do not neccessarily have $\sqsubseteq_{\mathrm{err}}$-limits.

$$[\![\mathbf{0}]\!] = 0$$

$$[\![\mathbf{A} + \mathbf{B}]\!] = [\![\mathbf{A}]\!] + [\![\mathbf{B}]\!]$$

$$[\![\mathbb{N}]\!] = \mathbb{N}$$

$$[\![\bigtimes_{\mathbf{i}\in\mathbf{I}}\mathbf{A}_\mathbf{i}]\!] = \prod_{\mathbf{i}\in\mathbf{I}} [\![\mathbf{A}_\mathbf{i}]\!]$$

$$[\![\prod_{\mathbf{i}\in\mathbf{I}}\mathbf{A}_\mathbf{i}]\!] = \{p \in \prod_{\mathbf{i}\in\mathbf{I}}[\![\mathbf{A}_\mathbf{i}]\!] | \mathrm{supp}(p) \text{is finite}\}[\![\mathbf{A} \to \mathbf{B}]\!] = U_{\Omega,\mho}([\![\mathbf{A}]\!] \to_{\Omega,\mho} F_{\Omega,\mho}[\![\mathbf{B}]\!])$$

$$[\![?]\!] = ?$$

$$[\![\mathbf{x}_1 : \mathbf{A}_1, \ldots, \mathbf{x}_n : \mathbf{A}_n]\!] = [\![\mathbf{A}_1]\!] \times \cdots \times [\![\mathbf{A}_n]\!]$$

A term $\mathbf{\Gamma} \vdash \mathbf{t} : \mathbf{A}$ denotes a call-by-value map $[\![\mathbf{t}]\!] : [\![\mathbf{\Gamma}]\!] \to_{\Omega,\mho} F_{\Omega,\mho}[\![\mathbf{A}]\!]$.

An evaluation context $\mathbf{E}[\cdot : \mathbf{A}] : \mathbf{B}$ denotes a doubly strict continuous function $[\![\mathbf{E}]\!] : F_{\Omega,\mho}[\![\mathbf{A}]\!] \to_{ss} F_{\Omega,\mho}[\![\mathbf{B}]\!]$, which is equivalently a call-by-value map $[\![\mathbf{A}]\!] \to_{\Omega,\mho} F_{\Omega,\mho}[\![\mathbf{B}]\!]$

A.2.3 Soundness and Adequacy.

LEMMA A.17 (COMPOSITIONALITY). *In summary, the semantics preserves every kind of composition in the source language*

(1) $\mathbf{t}[\mathbf{v}/\mathbf{x}] = [\![\mathbf{t}]\!] \circ_{\mathbf{x}} [\![\mathbf{v}]\!]$
(2) $\mathbf{E}[\mathbf{t}] = [\![\mathbf{E}]\!]([\![\mathbf{t}]\!])$
(3) $\mathbf{C}[\mathbf{t}] = [\![\mathbf{C}]\!]([\![\mathbf{t}]\!])$

THEOREM A.18 (SOUNDNESS OF DENOTATIONAL SEMANTICS). *If* $\mathbf{t} \longmapsto^* \mathbf{u}$ *then* $[\![\mathbf{t}]\!] = [\![\mathbf{u}]\!]$

PROOF. By compositionality, it suffices to prove the cases for an empty $\mathbf{E}$. All cases are direct from the universal properties of the types. □

THEOREM A.19 (ADEQUACY). *If* $\cdot \vdash \mathbf{t} : \mathbf{1}$, *then*

(1) $[\![\mathbf{t}]\!] = \lfloor d \rfloor$ *if and only if* $\mathbf{t} \longmapsto^* \langle\rangle$
(2) $[\![\mathbf{t}]\!] = \mho$ *if and only if* $\mathbf{t} \longmapsto^* \mho$
(3) $[\![\mathbf{t}]\!] = \Omega$ *if and only if* $\mathbf{t} \not\longmapsto^* \langle\rangle$ *and* $\mathbf{t} \not\longmapsto^* \mho$.

$$
\begin{array}{rcl}
\llbracket \mathbf{x}_1 : \mathbf{A}_1, \ldots, \mathbf{x}_n : \mathbf{A}_n \vdash \mathbf{t} : \mathbf{B} \rrbracket & : & \llbracket \mathbf{A}_1 \rrbracket \times \cdots \times \mathbf{A}_n \to_{\Omega,\mho} \llbracket \mathbf{B} \rrbracket \\
\llbracket \mathbf{x}_1 : \mathbf{A}_1, \ldots, \mathbf{x}_n : \mathbf{A}_n \vdash \mathbf{x}_i : \mathbf{A}_i \rrbracket(p) & = & \lfloor \pi_i(p) \rfloor \\
\llbracket \mho \rrbracket(p) & = & \mho \\
\llbracket \mathbf{roll}_? \, \mathbf{t} \rrbracket(p) & = & F_{\Omega,\mho} i(\llbracket \mathbf{t} \rrbracket(p)) \\
\llbracket \mathbf{unroll}_? \, \mathbf{t} \rrbracket(p) & = & F_{\Omega,\mho} i^{-1}(\llbracket \mathbf{t} \rrbracket(p)) \\
\llbracket \mathbf{rec}\, \mathbf{x}_f \, \mathbf{y} \,.\, \mathbf{t} \rrbracket(p) & = & Y(\lambda f.\lambda y.\llbracket \mathbf{t} \rrbracket(p, f, y)) \\
\llbracket \mathbf{n} \rrbracket(p) & = & \lfloor n \rfloor \\
\llbracket \mathbf{sub1}\, (\mathbf{t}) \rrbracket(p) & = & F_{\Omega,\mho}(\lambda x.x - 1)(\llbracket \mathbf{t} \rrbracket) \\
\llbracket \mathbf{case}\, \mathbf{t}\, \mathbf{of}\, () \rrbracket(p) & = & F_{\Omega,\mho}(¡) \mathrel{=\!\!\ll} (\llbracket \mathbf{t} \rrbracket(p)) \\
\llbracket \mathbf{case}\, \mathbf{t}\, \mathbf{of}\, \mathbf{x}_1 \,.\, \mathbf{t}_1 \mid \mathbf{x}_2 \,.\, \mathbf{t}_2 \rrbracket(p) & = & [\llbracket \mathbf{t}_1 \rrbracket(p, -), \llbracket \mathbf{t}_2 \rrbracket(p, -)] \mathrel{=\!\!\ll} \llbracket \mathbf{t} \rrbracket(p) \\
\llbracket \mathbf{inj}_\mathbf{i}\, \mathbf{t} \rrbracket p & = & F_{\Omega,\mho} \sigma_i(\llbracket \mathbf{t} \rrbracket(p)) \\
\llbracket \pi_\mathbf{i}\, \mathbf{t} \rrbracket(p) & = & F_{\Omega,\mho} \pi_\mathbf{i}(\llbracket \mathbf{t} \rrbracket(p)) \\
\llbracket \langle \mathbf{t}_{\mathbf{i}_1}, \ldots, \mathbf{t}_{\mathbf{i}_n} \rangle_\mathbf{I} \rrbracket(p) & = & F_{\Omega,\mho}(\lambda p.(\mathbf{i}_j \mapsto \pi_j p))(\mathrm{seq}(\llbracket \mathbf{t}_{\mathbf{i}_1} \rrbracket(p), \ldots, \llbracket \mathbf{t}_{\mathbf{i}_n} \rrbracket(p))) \\
\llbracket \mathbf{t.i} \rrbracket(p) & = & (\lfloor \circ \rfloor \pi_\mathbf{i}) \mathrel{=\!\!\ll} (\llbracket \mathbf{t} \rrbracket(p)) \\
\llbracket \lambda \mathbf{x}.\, \mathbf{t} \rrbracket(p) & = & \lfloor \lambda x.\llbracket \mathbf{t} \rrbracket(p, x) \rfloor \\
\llbracket \mathbf{t}\ \mathbf{u} \rrbracket(p) & = & (\lambda(f, x).f(x)) \mathrel{=\!\!\ll} \mathrm{seq}(\llbracket \mathbf{t} \rrbracket(p), \llbracket \mathbf{u} \rrbracket(p)) \\
\llbracket \mathbf{let}\, \mathbf{x} = \mathbf{t}\, \mathbf{in}\, \mathbf{u} \rrbracket(p) & = & \llbracket \mathbf{u} \rrbracket(p, -) \mathrel{=\!\!\ll} \llbracket \mathbf{t} \rrbracket(p) \\
\\
\mathrm{seq} & : & F_{\Omega,\mho} X_1 \times \cdots \times F_{\Omega,\mho} X_n \to F_{\Omega,\mho}(X_1 \times \cdots \times X_n) \\
\mathrm{seq}() & = & \lfloor () \rfloor \\
\mathrm{seq}(d_{car}, d_{cdr}) & = & (\lambda x.(\lambda y.(x, y) \mathrel{=\!\!\ll} d_{cdr})) \mathrel{=\!\!\ll} d_{car} \\
\mathrel{=\!\!\ll} & : & (X \to F_{\Omega,\mho} Y) \times F_{\Omega,\mho} X \to F_{\Omega,\mho} Y \\
f \mathrel{=\!\!\ll} \Omega & = & \Omega \\
f \mathrel{=\!\!\ll} \mho & = & \mho \\
f \mathrel{=\!\!\ll} \lfloor x \rfloor & = & f(x)
\end{array}
$$

Fig. 24. Denotational Semantics

Proof. The left implications of 1 and 2 follow from soundness. The right implications of 1 and 2 follow from the fundamental lemma. Then 3 follows from the first 2 and exhaustiveness. □

Theorem A.20 (Denotational Approximation implies Contextual Approximation). *For any* $\Gamma \vdash \mathbf{t} : \mathbf{A}$, $\Gamma \vdash \mathbf{u} : \mathbf{A}$, *if* $\llbracket \mathbf{t} \rrbracket \sqsubseteq_{err} \llbracket \mathbf{u} \rrbracket$ *then* $\mathbf{t} \sqsubseteq^{ctx}_{err} \mathbf{u}$.

Proof. Let $\mathbf{C}$ be an appropriate context. Then by compositionality, $\llbracket \mathbf{C}[\mathbf{t}] \rrbracket = \llbracket \mathbf{C} \rrbracket(\llbracket \mathbf{t} \rrbracket)$ and $\llbracket \mathbf{C}[\mathbf{u}] \rrbracket = \llbracket \mathbf{C} \rrbracket(\llbracket \mathbf{u} \rrbracket)$.

Then since $\llbracket \mathbf{t} \rrbracket \sqsubseteq_{\mathrm{err}} \llbracket \mathbf{u} \rrbracket$, by monotonicity, $\llbracket \mathbf{C}[\mathbf{t}] \rrbracket \sqsubseteq_{\mathrm{err}} \llbracket \mathbf{C}[\mathbf{u}] \rrbracket$. Then the result holds by adequacy. □

A.2.4 Relations for Adequacy. We will need two systems of relations to prove our results. The first is a logical relation between syntax and semantics, indexed by types, that we use to prove adequacy, described in this section. The second is a system of relations between predomains, indexed by type precision judgments, that we use to prove the gradual guarantee.

We will use the notation $A \twoheadrightarrow B$ for the type of relations between A and B that are in some sense "A less than B" in that they are downward-closed in an ordering on A and upper closed in an ordering on B. For the adequacy theorem this will mean the divergence order, whereas in the gradual guarantee theorem this will mean the error ordering.

In this section, we follow [20], but with some removed redundancy in presentation. In Pitts, there are actually 2 notions of a function preserving a relation, one used in the construction of the invariant relation for the adequacy proof that only uses a denotational function and another used in the adequacy proof which says when a pair of a term and a function preserve a relation. Here, we will simplify a bit by recognizing that the former notion is actually the same as the latter when the syntactic term is the identity function.

The recipe for invariant relations has 4 steps: define a notion of admissible relation, define preservation of admissible relations, define an action of functors on admissible relations, and show relations admit enough structure (inverse images and intersections) to use the Knaster-Tarksi fixed point theorem.

First we define our relations. These relations define when an element of a predomain (divergence domain) approximates a syntactic value (term). In practice since we have a call-by-value language, the only relations between a domain and a type are the lifts of relations on predomains.

Definition A.21 (Syntax-Semantics Admissible Relation).

(1) An admissible relation $R : X \nrightarrow \mathbf{A}$ between a predomain X and a type $\mathbf{A}$ is a subset $R \subseteq \mathrm{Val}(\mathbf{A}) \times |X|$ such that for each (closed) value $\mathbf{v} : \mathbf{A}$, $\{R(x, \mathbf{v}) | x \in |X|\} \subseteq |X|$ is downward closed in $\leq_X$ and closed under directed limits in $\leq_X$
(2) An admissible relation $R : Y \nrightarrow \mathbf{A}$ between a divergence domain Y and a type $\mathbf{A}$ is a subset $R \subseteq |X| \times \mathrm{Val}(\mathbf{A})$ such that for each (closed) value $\mathbf{v} : \mathbf{A}$, $\{R(x, \mathbf{v}) | x \in |X|\} \subseteq |X|$ contains Ω, is downward closed in $\leq_X$ and is closed under directed limits in $\leq_X$.
(3) The lift $R_{\Omega,\mho}$ of an admissible relation $R : X \nrightarrow \mathbf{A}$ where X is a predomain is an admissible relation $R_{\Omega,\mho} : U_\mho F_{\Omega,\mho} R \nrightarrow \mathbf{A}$ defined by

$$R_{\Omega,\mho}(x, \mathbf{t}) \overset{\text{def}}{=} \begin{cases} \top & \text{if } x = \Omega \\ \mathbf{t} \longmapsto^* \mho & \text{if } x = \mho \\ \mathbf{t} \longmapsto^* \mathbf{v} \wedge R(y, \mathbf{v}) & \text{if } x = \lfloor y \rfloor \end{cases}$$

Next, we say when a diverging,erroring function approximates a syntactic term, which is when they take related values to related computations.

Definition A.22 (Preservation of Relations). Given $\mathbf{x}_1 : \mathbf{A}_1, \ldots, \mathbf{x}_n : \mathbf{A}_n \vdash \mathbf{t} : \mathbf{B}$, $f : X_1 \times \cdots \times X_n \to_{\Omega,\mho} Y$ where $X_1, \ldots, X_n$ are predomains and Y is a divergence domains, and admissible relations $R_1 : X_1 \nrightarrow \mathbf{A}_1, \cdots, R_n : X_n \nrightarrow \mathbf{A}_n, \cdots, S : Y \nrightarrow \mathbf{B}$, we say f formally approximates $\mathbf{t}$ at relations $R_1, \ldots, R_n, S$, written $\mathbf{x}_1 : R_1, \ldots, \mathbf{x}_n : R_n \vDash f \leq \mathbf{t} \in S$ if for any $R_1(x_1, \mathbf{v}_1), \ldots, R_n(x_n, \mathbf{v}_n)$, $S_{\Omega,\mho}(f(x_1, \ldots, x_n), \mathbf{t}[\gamma])$ where $\gamma(\mathbf{x}_i) = \mathbf{v}_i$.

Note that in the case where $\mathbf{t} = \mathbf{x}$, this is almost exactly the same as the definition of preservation of relations given in [20] for the relational structure used in the adequacy proof, but here we maintain the typing distinction for the syntactic values.

We can prove inclusion of relations using preservation using the following lemma:

Lemma A.23 (Inclusion is Preservation by Identity). *For $R, R' : X \nrightarrow \mathbf{A}$*

$$R \subseteq R' \quad \text{iff} \quad \mathbf{x} : R \vDash \lambda x.\lfloor x \rfloor \leq \mathbf{x} \in R'$$

Lemma A.24 (Reduction preserves and reflects Relatedness). *If $\mathbf{t} \longmapsto \mathbf{u}$ then $(d, \mathbf{t}) \in R_{\Omega,\mho}iff(d, \mathbf{u}) \in R_{\Omega,\mho}$*

Proof. Immediate by determinacy of reduction. □

In particular this gives us the identity rule $\mathbf{x} : R \vDash \mathrm{id} \leq \mathbf{x} \in R$. We also have a composition rule:

Lemma A.25 (Composition of Related Function-Term Pairs). *If $\mathbf{x}_1 : R_1, \ldots, \mathbf{x}_n : R_n \vDash f \leq \mathbf{t} \in S$ and $\mathbf{x}_1 : R_1, \ldots, \mathbf{x}_n : R_n, \mathbf{y} : S \vDash g \leq \mathbf{u} \in T$ then $\mathbf{x}_1 : R_1, \ldots, \mathbf{x}_n : R_n \vDash \lambda(p).g(p, f(p)) \leq \mathbf{let}\ \mathbf{y} = \mathbf{t}\ \mathbf{in}\ \mathbf{u} \in T$*

Proof. Given p, γ, if $f(p)$ diverges, so does $g(p, f(p))$, so we're done. If $f(p)$ errors, so does $\mathbf{t}[\gamma]$ so both sides error.

Finally, if $f(p) = \lfloor d \rfloor$, then $\mathbf{t}$ reduces to a related value which is substituted into $\mathbf{u}$ and the result follows from relatedness of $g, \mathbf{u}$. □

Usually we won't have an explicit "let" in a term, but we essentially have one if the term $\mathbf{u}$ is *strictly linear*:

Definition A.26 (Strictly Linear Term). A term $\mathbf{t}$ is strictly linear in $\mathbf{x}$ is $\mathbf{t} = \mathbf{E}[\mathbf{x}]$ for some evaluation context $\mathbf{E}$.

Lemma A.27. *If* $\mathbf{t}$ *is strictly linear in* $\mathbf{x}$ *than* $\Gamma \vDash f \leq \mathbf{let\,x = u\,in\,t} \in R$ *if and only if* $\Gamma \vDash f \leq \mathbf{t}[\mathbf{u}/\mathbf{x}] \in R$.

This is often used when $\mathbf{t} = \mathrm{id}$ and we want to decompose the denotational function.

Finally, we also need an induction principle:

Lemma A.28 (Relatedness Induction). *If* f *is the least fixed point of a directed set* D *and for every* $g \in D, \Gamma \vDash g \leq \mathbf{t} \in R$, *then* $\Gamma \vDash f \leq \mathbf{t} \in R$

Proof. Let p, γ related, then we need to show $R_{\Omega,\mho}(f(p), \mathbf{t}[\gamma])$. Since application is continuous, $f(p) = \bigvee\{g(p) | g \in D\}$. Then the result follows by admissibility of R. □

Next, we extend the locally continuous functors defined in [20] to actions on admissible relations.

Here, we stick closely to Pitts' definition by only considering identities on the syntactic side. Morally this would require the type constructor $\mathbf{F}$ to be functorial, but this presents a bootstrapping issue since we are not yet equipped to prove any equalities between terms. Instead, since $\mathbf{F}$ *ought* to be functorial then in particular $\mathbf{F}(\mathrm{id}, \ldots, \mathrm{id})$ *ought* to be equal to just id.

Another detail to note is that locally continuous functors are defined on PreDom_Ω, but preservation of relations is defined on morphisms in $\mathrm{PreDom}_{\Omega,\mho}$, but there is a functor $\mathrm{PreDom}_\Omega \to \mathrm{PreDom}_{\Omega,\mho}$ that composes $f : X \to F_\Omega Y$ with the "error lift" $F_\Omega Y \to U_\mho F_{\Omega,\mho}$. We leave this promotion implicit in the following definition for simplicity's sake.

Definition A.29 (Action of a Functor on Relations). Given a locally continuous functor $F : (\mathrm{PreDom}_\Omega^{op})^n \times \mathrm{PreDom}_\Omega^m \to \mathrm{PreDom}_\Omega$ and a type constructor $\mathbf{F} : \mathrm{Type}^{n+m} \to \mathrm{Type}$, an admissible action of $F, \mathbf{F}$ is a function

$$F : \Pi_{i=1}^n (X_i \nrightarrow \mathbf{A}_i) \times \Pi_{j=1}^m (Y_j \nrightarrow \mathbf{B}_i) \to F(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \nrightarrow \mathbf{F}(\mathbf{A}_1, \ldots, \mathbf{A}_n, \mathbf{B}_1, \ldots, \mathbf{B}_n)$$

such that given $\mathbf{x} : R_i' \vDash f_i \leq \mathbf{x} \in R_i$ for $i = 1, \ldots, n$ and $\mathbf{x} : S_j \vDash g_j \leq \mathbf{x} \in S_i j'$ for $j = 1, \ldots, m$, we have

$$\mathbf{x} : F(R_1, \ldots, R_n, S_1, \ldots, S_m) \vDash F(f_1, \ldots, f_n, g_1, \ldots, g_m) \leq \mathbf{x} \in F(R_1', \ldots, R_n', S_1', \ldots, S_m')$$

Definition A.30 (Action of Denotations on Relations). The actions of relations defined in Figure 25 are all admissible actions.

Proof. All cases are straightforward, composition is used for the function case. □

Finally, we show that admissible relations admit intersections and inverse images. Note that we need these for relations on predomains and values, not predomains and values.

Intersection is straightforward:

$$\begin{array}{rcl}
+ & : & (X \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A}) \times (Y \mathrel{\mapstochar\twoheadrightarrow} \mathbf{B}) \to (X + Y \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A} + \mathbf{B}) \\
(R + Q)(d, \mathbf{v}) & \stackrel{\text{def}}{=} & ((d = \sigma_1 d_1 \wedge \mathbf{v} = \mathbf{inj_1}\, \mathbf{v}_1 \wedge R(d_1, \mathbf{v}_1)) \vee (d = \sigma_2 d_2 \wedge \mathbf{v} = \mathbf{inj_2}\, \mathbf{v}_2 \wedge S(d_2, \mathbf{v}_2))) \\
0 & : & 1 \to (0 \mathrel{\mapstochar\twoheadrightarrow} \mathbf{0}) \\
0(d, \mathbf{v}) & \stackrel{\text{def}}{=} & \bot \\
\mathbb{N} & : & 1 \to (\mathbb{N} \mathrel{\mapstochar\twoheadrightarrow} \mathbb{N}) \\
\mathbb{N}(n, \mathbf{m}) & \stackrel{\text{def}}{=} & n = \mathbf{m} \\
\to & : & (X \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A})^{op} \times (Y \mathrel{\mapstochar\twoheadrightarrow} \mathbf{B}) \to ((X \to U_{\Omega} F_{\Omega,\mho} Y) \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A} \to \mathbf{B}) \\
(R \to S)(f, \mathbf{v}) & \stackrel{\text{def}}{=} & \mathbf{x} : R \vDash f \leq \mathbf{v}\, \mathbf{x} \in S \\
\Pi_{\mathbf{i} \in \mathbf{I}} & : & (X_{\mathbf{i}_1} \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A}_1) \times \cdots \times (X_{\mathbf{i}_{|\mathbf{I}|}} \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A}_{|\mathbf{I}|}) \to \Pi_{\mathbf{i} \in \mathbf{I}} X_{\mathbf{i}} \mathrel{\mapstochar\twoheadrightarrow} \bigtimes_{\mathbf{i} \in \mathbf{I}} \mathbf{A}_{\mathbf{i}} \\
\Pi_{\mathbf{i} \in \mathbf{I}} R_{\mathbf{i}}(d, \langle \ldots, \mathbf{v}_{\mathbf{i}}, \ldots \rangle_{\mathbf{i} \in \mathbf{I}}) & \stackrel{\text{def}}{=} & \forall \mathbf{i} \in \mathbf{I}.\ R_{\mathbf{i}}(\pi_{\mathbf{i}} d, \mathbf{v}_{\mathbf{i}})
\end{array}$$

Fig. 25. admissible actions on relations

Definition A.31 (Intersection of Admissible Relations). Let $\mathfrak{R} \subset X \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A}$ for some predomain X, type $\mathbf{A}$. Then the intersection $\bigcap \mathfrak{R} : X \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A}$ is defined by

$$\bigcap \mathfrak{R}(x, \mathbf{t}) = \forall R \in \mathfrak{R}, R(x, \mathbf{t})$$

Which is admissible if all the $R \in \mathfrak{R}$ are.

This clearly has the property that

$$\mathbf{x}_1 : R_1, \ldots, \mathbf{x}_n : R_n \vDash g \leq \mathbf{u} \in \bigcap \mathfrak{R} \quad \text{iff} \quad \forall R \in \mathfrak{R}. \mathbf{x}_1 : R_1, \ldots, \mathbf{x}_n : R_n \vDash g \leq \mathbf{u} \in R$$

Inverse images can be defined as well, specifically we will define the inverse image along a continuous function and a “pure” term. For this we can get away with an overly simplistic syntactic notion of purity.

Definition A.32 (Pure Term). A term $\mathbf{x} : \mathbf{A} \vdash \mathbf{t} : \mathbf{B}$ is *pure* when it is strictly linear and for any closed $\mathbf{v} : \mathbf{A}$, there exists a unique $\mathbf{v}' : \mathbf{B}$ such that $\mathbf{E}[\mathbf{v}] \longmapsto^* \mathbf{v}'$.

Of course the uniqueness here is free since evaluation is deterministic. For the adequacy proof, we will only need to instantiate this definition with the syntactic $\mathbf{?}$ recursive type isomorphism.

Definition A.33 (Inverse Image of a Relation). Given an admissible relation $R : Y \mathrel{\mapstochar\twoheadrightarrow} \mathbf{B}$, a continuous function of predomains $f : X \to Y$ and a pure term $\mathbf{x} : \mathbf{A} \vdash \mathbf{t} : \mathbf{B}$, we define the *inverse image* $(f, \mathbf{x}.\mathbf{t})^* R : X \mathrel{\mapstochar\twoheadrightarrow} \mathbf{A}$ by

$$((f, \mathbf{x}.\mathbf{t})^* R)(d, \mathbf{v}) \stackrel{\text{def}}{=} R(f(d), \mathbf{v}')$$

where $\mathbf{t}[\mathbf{v}] \longmapsto^* \mathbf{v}'$

This can be verified to be admissible using the fact that f is continuous.

We will use this definition only by the following reasoning principles.

Lemma A.34 (Reasoning Principles for Inverse Images).

$$\mathbf{x}_1 : R_1, \ldots, \mathbf{x}_n : R_n \vDash g \leq \mathbf{u} \in (f, \mathbf{y}.\mathbf{t})^* S \quad \text{iff} \quad \mathbf{x}_1 : R_1, \ldots, \mathbf{x}_n : R_n \vDash f \circ g \leq \mathbf{t}[\mathbf{u}/\mathbf{y}] \in S$$

and

$$\mathbf{x} : (f, \mathbf{t})^* R \vDash f \leq \mathbf{t} \in R$$

Proof. Essentially by definition, using the fact that $f, \mathbf{t}$ are pure. □

Finally we can prove existence of invariant relations that are “over” a minimal invariant on the semantic side and an “obvious” isomorphism on the syntactic side.

Definition A.35 (Obvious Isomorphism). An obvious isomorphism from $\mathbf{A}$ to $\mathbf{B}$ is a pair of pure terms $\mathbf{x} : \mathbf{A} \vdash \mathbf{t} : \mathbf{B}, \mathbf{y} : \mathbf{B} \vdash \mathbf{t}^{-1} : \mathbf{A}$ such that for any closed $\mathbf{v} : \mathbf{A}$,

$$\mathbf{t}^{-1}[\mathbf{t}[\mathbf{v}/\mathbf{x}]/\mathbf{y}] \longmapsto^* \mathbf{v}$$

and for any closed $\mathbf{v} : \mathbf{B}$

$$\mathbf{t}[\mathbf{t}^{-1}[\mathbf{v}/\mathbf{y}]/\mathbf{x}] \longmapsto^* \mathbf{v}$$

Theorem A.36 (Invariant Relations Exist). *For any locally continuous functor $F : (PreDom_\Omega^{op})^1 \times PreDom_\Omega^1 \to PreDom_\Omega$ with an action on relations tracked by the type constructor $\mathbf{F}$, given a minimal invariant $i : F(X, X) \cong X$ and an obvious isomorphism $\mathbf{t} : \mathbf{FAA} \cong \mathbf{A}$, then there exists an invariant admissible relation $R_F : \mathbf{A} \nrightarrow X$, that is*

$$\mathbf{x} : R_F \vDash i^{-1} \leq \mathbf{x}.\mathbf{t}^{-1} \in F(R_F, R_F) \quad \textit{and} \quad \mathbf{x} : F(R_F, R_F) \vDash i \leq \mathbf{x}.\mathbf{t} \in R_F$$

Proof. Following Pitts, but adapting to our modified definitions.

First, R_F satisfies our definition if it is a fixed point of the function $\mathrm{grow} : (X \nrightarrow \mathbf{A}) \to (X \nrightarrow \mathbf{A})$ defined by

$$\mathrm{grow}(R) = (\mathbf{t}^{-1}, i^{-1})^* F(R, R)$$

however grow is not monotone, so we separate positive and negative occurences construct the monotone $\mathrm{grow}' : (X \nrightarrow \mathbf{A})^{op} \times (X \nrightarrow \mathbf{A}) to (X \nrightarrow \mathbf{A})$ defined by

$$\mathrm{grow}'(R^-, R^+) = (\mathbf{t}^{-1}, i^{-1})^* F(R^-, R^+)$$

Then we symmetrize it make it an endofunction $\mathrm{grow}'^{\S}(R^-, R^+) = (\mathrm{grow}'(R^+, R^-), \mathrm{grow}'(R^-, R^+))$. Next, we can use the Knaster-Tarski fixed point theorem because admissible relations for fixed types form a complete lattice since they admit arbitrary meets (joins can be defined by infinite meets). Giving us R_F^-, R_F^+ satisfying

$$\mathrm{grow}'(R_F^-, R_F^+) = R_F^+ \quad \text{and} \quad \mathrm{grow}'(R_F^+, R_F^-) = R_F^-$$

and unrolling definitions:

$$\forall R^-, R^+, if (R^- \subset \mathrm{grow}'(R^+, R^-)) \text{and} (\mathrm{grow}'(R^-, R^+) \subseteq R^+) \text{ then } R^- \subseteq R_F^- \wedge R_F^+ \subseteq R^+$$

So we are finished if $R_F^+ = R_F^-$. We have $R_F^+ \subseteq R_F^-$ from the above, so we need to show $R_F^- \subseteq R_F^+$, equivalently that $\mathbf{x} : R_F^- \vDash \mathrm{id} \leq \mathbf{x} \in R^+_{F\Omega,\mho}$. To prove this we use the minimal invariance property of i, we know id is the least fixed point of $\delta(e) = i \circ F(e, e) \circ i^{-1}$.

First by definition of ${}_{\Omega,\mho}$ and the definition of obvious isomorphism, $\mathbf{x} : R_F^- \vDash \mathrm{id} \leq \mathbf{x} \in R^+_{F\Omega,\mho}$ holds if and only if $\mathbf{x} : R_F^- \vDash \mathrm{id} \leq \mathbf{t}[\mathbf{t}^{-1}[\mathbf{x}]] \in R^+_{F\Omega,\mho}$.

Next, by the minimal invariance of i, we can proceed by induction. It is sufficient to show

$$\mathbf{x} : R_F^- \vDash \Omega \leq \mathbf{t}[\mathbf{t}^{-1}[\mathbf{x}]] \in R^+_{F\Omega,\mho}$$

which holds trivially, and for any f,

$$\mathbf{x} : R_F^- \vDash f \leq \mathbf{t}[\mathbf{t}^{-1}[\mathbf{x}]] \in R^+_{F\Omega,\mho} \implies \mathbf{x} : R_F^- \vDash i \circ F(f, f) \circ i^{-1} \leq \mathbf{t}[\mathbf{t}^{-1}[\mathbf{x}]] \in R^+_{F\Omega,\mho}$$

which follows from the following three cases:

(1) $\mathbf{x} : F(R_F^-, R_F^+) \vDash i^{-1} \leq \mathbf{t}^{-1}[\mathbf{x}] \in R_F^+$: immediate from $R_F^+ = (\mathbf{t}^{-1}, i^{-1})^* F(R_F^-, R_F^+)$ and definition of inverse images.

(2) $\mathbf{x} : F(R_F^+, R_F^-) \vDash F(f, f) \leq \mathbf{x} \in F(R_F^-, R_F^+)$: immediate by definition of admissible action of F.

(3) $\mathbf{x} : R_F^- \vDash i \le \mathbf{t}[\mathbf{x}] \in F(R_F^+, R_F^-)$: since $R_F^- = (\mathbf{t}^{-1}, i^{-1})^* F(R_F^+, R_F^-)$ and by Lemma A.54 it is sufficient to show

$$\mathbf{x} : F(R_F^+, R_F^-) \vDash ii^{-1} \le \mathbf{t}[\mathbf{u}[\mathbf{x}]] \in F(R_F^+, R_F^-)$$

which follows from $ii^{-1} = \mathrm{id}$ and $\mathbf{t}[\mathbf{u}[\mathbf{v}]] \longmapsto^* \mathbf{v}$.

□

Lemma A.37 (Fundamental Lemma). *If* $\mathbf{x}_1 : \mathbf{A}_1, \dots, \mathbf{x}_n : \mathbf{A}_n \vdash \mathbf{t} : \mathbf{B}$, *then*

$$\mathbf{x}_1 : \mathcal{V}\llbracket \mathbf{A}_1 \rrbracket, \dots, \mathbf{x}_n : \mathcal{V}\llbracket \mathbf{A}_n \rrbracket \vDash \llbracket \mathbf{t} \rrbracket \le \mathbf{t} \in \mathcal{V}\llbracket \mathbf{B} \rrbracket$$

Proof. By induction on the derivation of $\mathbf{x}_1 : \mathbf{A}_1, \dots, \mathbf{x}_n : \mathbf{A}_n \vdash \mathbf{t} : \mathbf{B}$. All cases follow easily, the only interesting case is recursion:

(1) We need to show $\mathbf{x}_1 : \mathcal{V}\llbracket \mathbf{A}_1 \rrbracket, \dots, \mathbf{x}_n : \mathcal{V}\llbracket \mathbf{A}_n \rrbracket \vDash \lambda p. Y(\lambda x. \llbracket mterm \rrbracket(p, x))^n(\bot) \le \mathbf{rec}\, \mathbf{x}_f\, \mathbf{y} \,.\, \mathbf{t} \in \mathcal{V}\llbracket \mathbf{B} \rrbracket$.

Assume $\mathcal{V}\llbracket \mathbf{A}_1 \rrbracket(x_1, \mathbf{v}_1), \dots, \mathcal{V}\llbracket \mathbf{A}_n \rrbracket(x_n, \mathbf{v}_n)$, define $\gamma(\mathbf{x}_i) = \mathbf{v}_i$ and $p = (x_1, \dots, x_n)$. We need to show that

$$\mathcal{V}\llbracket \mathbf{B} \to \mathbf{B}' \rrbracket_{\Omega,\mho}(\lfloor Y(g) \rfloor, \mathbf{rec}\, \mathbf{x}_f\, \mathbf{y} \,.\, \mathbf{t}[\gamma])$$

where $g(f) = \lambda y. \llbracket \mathbf{t} \rrbracket(p, f, y)$ Since the semantic side is a lift and the syntactic side is a value, it is sufficient to show that

$$\mathbf{y} : \mathcal{V}\llbracket \mathbf{B} \rrbracket \vDash Y(g) \le (\mathbf{rec}\, \mathbf{x}_f\, \mathbf{y} \,.\, \mathbf{t}[\gamma])\, \mathbf{y} \in \mathcal{V}\llbracket \mathbf{B}' \rrbracket$$

By admissibility we can proceed by fixed point induction. The Ω case is trivial. Assume $f : \llbracket \Gamma \rrbracket \to F_{\Omega,\mho} A \to B$ with

$$\mathbf{y} : \mathcal{V}\llbracket \mathbf{B} \rrbracket \vDash f \le (\mathbf{rec}\, \mathbf{x}_f\, \mathbf{y} \,.\, \mathbf{t}[\gamma])\, \mathbf{y} \in \mathcal{V}\llbracket \mathbf{B}' \rrbracket$$

we need to show that

$$\mathbf{y} : \mathcal{V}\llbracket \mathbf{B} \rrbracket \vDash g(f) \le (\mathbf{rec}\, \mathbf{x}_f\, \mathbf{y} \,.\, \mathbf{t}[\gamma])\, \mathbf{y} \in \mathcal{V}\llbracket \mathbf{B}' \rrbracket$$

that is

$$\mathbf{y} : \mathcal{V}\llbracket \mathbf{B} \rrbracket \vDash \lambda y. \llbracket \mathbf{t} \rrbracket(p, f, y) \le (\mathbf{rec}\, \mathbf{x}_f\, \mathbf{y} \,.\, \mathbf{t}[\gamma])\, \mathbf{y} \in \mathcal{V}\llbracket \mathbf{B}' \rrbracket$$

Given some $\mathcal{V}\llbracket \mathbf{B} \rrbracket(d, \mathbf{v})$, we need to show that

$$\mathcal{V}\llbracket \mathbf{B}' \rrbracket_{\Omega,\mho}(\llbracket \mathbf{t} \rrbracket(p, f, d), (\mathbf{rec}\, \mathbf{x}_f\, \mathbf{y} \,.\, \mathbf{t}[\gamma])\, \mathbf{v})$$

By Lemma A.24, this is equivalent to

$$\mathcal{V}\llbracket \mathbf{B}' \rrbracket_{\Omega,\mho}(\llbracket \mathbf{t} \rrbracket(p, f, d), \mathbf{t}[\mathbf{rec}\, \mathbf{x}_f\, \mathbf{y} \,.\, \mathbf{t}[\gamma]/\mathbf{x}_f][\mathbf{v}/\mathbf{y}])$$

Which follows from our inductive hypotheses about $\mathbf{t}$ and f.

□

$$
\begin{array}{rcl}
\mathcal{V}\llbracket \mathbf{A} \rrbracket & : & \llbracket \mathbf{A} \rrbracket \nrightarrow \mathbf{A} \\
\mathcal{V}\llbracket ? \rrbracket (d, \mathbf{roll}_? \, \mathbf{v}) & \text{iff} & \mathcal{V}\left\llbracket \mathbf{1} + \mathbb{N} + (? + ?) + (? \times ?) + (? \to ?) + \prod_{- \in \mathfrak{F}} ? \right\rrbracket (i^{-1}(d), \mathbf{v}) \\
\mathcal{V}\llbracket \bigtimes_{\mathbf{i} \in \mathbf{I}} \mathbf{A_i} \rrbracket (d, \langle \ldots, \mathbf{v_i}, \ldots \rangle_{\mathbf{i} \in \mathbf{I}}) & \text{iff} & \forall \mathbf{i} \in \mathbf{I}.\ \mathcal{V}\llbracket \mathbf{A_i} \rrbracket (\pi_{\mathbf{i}} d, \mathbf{v_i}) \\
\mathcal{V}\llbracket \mathbf{A_1} + \mathbf{A_2} \rrbracket (d, \mathbf{v}) & \text{iff} & (d = \sigma_1 d_1 \wedge \mathbf{v} = \mathbf{inj_1}\, \mathbf{v_1} \wedge \mathcal{V}\llbracket \mathbf{A_1} \rrbracket (d_1, \mathbf{v_1})) \vee (d = \sigma_2 d_2 \wedge \mathbf{v} = \mathbf{inj_2}\, \mathbf{v_2} \wedge \mathcal{V}\llbracket \mathbf{A_2} \rrbracket (\\
\mathcal{V}\llbracket \mathbf{0} \rrbracket (d, \mathbf{v}) & \text{iff} & \bot \\
\mathcal{V}\llbracket \mathbb{N} \rrbracket (n, \mathbf{m}) & \text{iff} & n = m \\
\mathcal{V}\llbracket \mathbf{A} \to \mathbf{B} \rrbracket (f, \mathbf{v}) & \text{iff} & \forall \mathcal{V}\llbracket \mathbf{A} \rrbracket (d, \mathbf{v'}).\ \mathcal{E}\llbracket \mathbf{B} \rrbracket (f(d), \mathbf{v}\, \mathbf{v'}) \\
\mathcal{E}\llbracket \mathbf{A} \rrbracket & : & F_{\Omega, \mho}\llbracket \mathbf{A} \rrbracket \nrightarrow \mathbf{A} \\
\mathcal{E}\llbracket \mathbf{A} \rrbracket (c, \mathbf{t}) & \text{iff} & \text{if } c = \mho, \mathbf{t} \longmapsto^* \mho \wedge \text{if } c = \lfloor d \rfloor, \exists \mathbf{v}.\ \mathbf{t} \longmapsto^* \mathbf{v} \wedge \mathcal{V}\llbracket \mathbf{A} \rrbracket (d, \mathbf{v})
\end{array}
$$

Fig. 26. Logical Relation for Adequacy (User-Friendly)

$$
\begin{array}{rcl}
\mathcal{V}\llbracket \mathbf{A} \rrbracket & : & \llbracket \mathbf{A} \rrbracket \nrightarrow \mathbf{A} \\
\mathcal{V}\llbracket ? \rrbracket & \overset{\text{def}}{=} & \text{invariant relation of Dyn} \\
\mathcal{V}\llbracket \bigtimes_{\mathbf{i} \in \mathbf{I}} \mathbf{A_i} \rrbracket & \overset{\text{def}}{=} & \Pi_{\mathbf{i} \in \mathbf{I}} (\mathcal{V}\llbracket \mathbf{A_i} \rrbracket (\mathcal{V}\llbracket ? \rrbracket, \mathcal{V}\llbracket ? \rrbracket)) \\
\mathcal{V}\llbracket \mathbf{A_1} + \mathbf{A_2} \rrbracket & \overset{\text{def}}{=} & \mathcal{V}\llbracket \mathbf{A_1} \rrbracket (\mathcal{V}\llbracket ? \rrbracket, \mathcal{V}\llbracket ? \rrbracket) + \mathcal{V}\llbracket \mathbf{A_2} \rrbracket (\mathcal{V}\llbracket ? \rrbracket, \mathcal{V}\llbracket ? \rrbracket) \\
\mathcal{V}\llbracket \mathbf{0} \rrbracket & \overset{\text{def}}{=} & 0(\cdot) \\
\mathcal{V}\llbracket \mathbb{N} \rrbracket & \overset{\text{def}}{=} & \mathbb{N}(\cdot) \\
\mathcal{V}\llbracket \mathbf{A} \to \mathbf{B} \rrbracket & \overset{\text{def}}{=} & \mathcal{V}\llbracket \mathbf{A} \rrbracket (\mathcal{V}\llbracket ? \rrbracket, \mathcal{V}\llbracket ? \rrbracket) \to \mathcal{V}\llbracket \mathbf{B} \rrbracket (\mathcal{V}\llbracket ? \rrbracket, \mathcal{V}\llbracket ? \rrbracket) \\
\mathcal{E}\llbracket \mathbf{A} \rrbracket & : & F_{\Omega, \mho}\llbracket \mathbf{A} \rrbracket \nrightarrow \mathbf{A} \\
\mathcal{E}\llbracket \mathbf{A} \rrbracket & \overset{\text{def}}{=} & \mathcal{V}\llbracket \mathbf{A} \rrbracket_{\Omega, \mho}
\end{array}
$$

Fig. 27. Logical Relation for Adequacy (Definition)

A.2.5 Error Embedding-Projection Pairs. Next, we prove that gradual types as implemented *are* embedding projection pairs as claimed.

Whereas in the previous section our primary interest was the category of possibly diverging functions, here our interest will be in the possibly erroring functions.

Definition A.38 (Error Domains). An error domain X is presented as a tuple $(|X|, \leq_X, \sqsubseteq_X, \mho_X)$ such that $(|X|, \leq_X, \sqsubseteq_X)$ is a predomain and $\mho_X \in |X|$ is the $\sqsubseteq_X$-least element. A strict continuous function $X \to_s Y$ of error domains is a continuous function of the underlying predomains that preserves $\mho$.

The error domains and strict continuous functions assemble into a category errDom and there is a monadic adjunction $F_\mho \dashv U_\mho$ between PreDom and errDom where $F_\mho$ freely adjoins an $\mho$ and $U_\mho$ takes the underlying predomain.

The category of possibly diverging functions $\text{PreDom}_\mho$ is the kleisli category of $F_\mho \dashv U_\mho$ which has predomains as objects and morphisms $X \to_\mho Y$ are equivalently presented by continuous functions $X \to U_\mho F_\mho Y$ or strict continuous functions $F_\mho X \to_s F_\mho Y$.

We will consider $\text{PreDom}_\mho$ as a 2-category with a unique 2-cell between any $f, g : X \to_\mho Y$ when $\forall x \in X. f(x) \sqsubseteq_{\text{err}} g(x)$, which we write $f \sqsubseteq_{\text{err}} g$. It will also be convenient at times to view $\text{PreDom}_\mho$ as a subcategory of $\text{PreDom}_{\Omega,\mho}$ of those functions that never result in Ω.

First, we clarify the semantic definition of embedding-projection pair.

Definition A.39 (Embedding Projection Pair). An *error embedding-projection pair* from a predomain X to a predomain Y is equivalently

(1) An adjunction in $\text{PreDom}_\mho$ for which the right adjoint is a retraction of the left adjoint.
(2) Two morphisms in $\text{PreDom}_\mho$, $e : X \to_\mho Y, p : Y \to_\mho X$ such that $p \circ e = \text{id}_X$ and $e \circ p \sqsubseteq_{\text{err}} \text{id}_Y$.

(3) Two morphisms in $\mathrm{PreDom}_{\Omega,\mho}$, $e : X \to_{\Omega,\mho} Y, p : Y \to_{\Omega,\mho} X$ such that $p \circ e = \mathrm{id}_X$ and $e \circ p \sqsubseteq_{\mathrm{err}} \mathrm{id}_Y$.
(4) A continuous function $e : X \to Y$ and a diverging-erroring $p : Y \to_{\Omega,\mho} X$ such that $p \circ e = \mathrm{id}_X$ and $e \circ p \sqsubseteq_{\mathrm{err}} \mathrm{id}_Y$.

Proof. Any $e : X \to_{\Omega,\mho} Y$ that has an inverse $p : Y \to_{\Omega,\mho} X$ must be pure because if $e(x) = \mho$ than $p(e(x)) = \mho \neq x$ which is a contradiction (similarly for divergence).

Furthermore any $p : Y \to_{\Omega,\mho} X$ can never error because $e(p(y)) = e(\Omega) = \Omega \not\sqsubseteq_{\mathrm{err}} \lfloor y \rfloor$ which is a contradiction. □

To construct our ep pairs, the following general observation is useful:

Lemma A.40 (All functors preserve EP Pairs). *Any 2-functor $F : C \to D$ of 2-categories preserves adjunctions and retractions.*

Proof. Since F preserves all compositions and 2-cells and adjunctions and retractions are defined purely as such equations. □

Specializing to our setting, the appropriate notion of functor is

Definition A.41 (Locally Monotone Functor). A functor $F : (\mathrm{PreDom}_{\mho}^{op})^n \times \mathrm{PreDom}_{\mho}^m \to \mathrm{PreDom}_{\mho}$ is locally monotone if for all $f_i \sqsubseteq_{\mathrm{err}} f_i'$ and $g_i \sqsubseteq_{\mathrm{err}} g_i'$, $F(f_1, \ldots, f_n, g_1 \ldots, g_n) \sqsubseteq_{\mathrm{err}} F(f_1', \ldots, f_n', g_1' \ldots, g_n')$.

The semantics of every type constructor defines a locally monotone functor using almost exactly the same definitions as given for locally continuity, but replacing $\leq_{\mathrm{div}}$ with $\sqsubseteq_{\mathrm{err}}$ appropriately.

In particular we use the fact that $U_{\mho}F_{\mho}$ is a commutative monad on PreDom.

We note here that functoriality depends on the fact that there is only one notion of "undefined" in the category we are considering. In particular the product does not naturally extend to a doubly monotone functor on $\mathrm{PreDom}_{\Omega,\mho}$.

Now we can prove our soundness theorems for gradual types and type precision. We prove them for the extended language under the assumption that the user-defined ep pairs are *semantic* ep pairs. Note that there is no circularity in the following condition because the definition of the elaboration process does not depend on well-formedness.

Definition A.42 (Semantically Well-Formed Syntax). A syntactic object (type, typing derivation, type precision judgment or term precision judgment) S in the extended surface language is well formed if for any occurrence of $\mathsf{Cast}\langle \mathsf{d} :: \mathsf{B}' \sqsubseteq \mathsf{B}; \mathsf{v}_e, \mathsf{v}_p : \mathsf{B} \triangleleft \mathsf{C}\rangle$ in S, $[\![\mathsf{v}_e]\!] : [\![\mathsf{B}]\!] \to_{\Omega,\mho} [\![\mathsf{C}]\!], [\![\mathsf{v}_p]\!] : [\![\mathsf{C}]\!] \to_{\Omega,\mho} [\![\mathsf{B}]\!]$ form an embedding-projection pair.

Theorem A.43 (Type Precision Judgments are EP Pairs). *For any well-formed derivation $d :: \mathsf{A} \sqsubseteq \mathsf{B}$, $[\![d^{\mathsf{e}}]\!], [\![d^{\mathsf{p}}]\!]$ form an embedding-projection pair from A to B.*

Proof. By induction on d.

(1) Tagging: $\mathsf{G} \sqsubseteq ?$, by inspection of definition.
(2) Identity, cut: from compositionality of semantics and that ep pairs compose.
(3) Cast refinements right rule: $[\![(\mathsf{Cast}\langle \mathsf{d}' :: \mathsf{A}' \sqsubseteq \mathsf{A}; \mathsf{v}_e, \mathsf{v}_p : \mathsf{A} \triangleleft \mathsf{B}\rangle \sqsubseteq \mathsf{B})^{\mathsf{e}}]\!] = [\![\mathsf{v}_e]\!]$ immediate by assumption that the derivation is well-formed.
(4) Cast refinements left rule: $[\![(\mathsf{Cast}\langle \mathsf{d}'' :: \mathsf{A}'' \sqsubseteq \mathsf{A}; \mathsf{v}, \mathsf{v}' : \mathsf{A} \triangleleft \mathsf{B}\rangle \sqsubseteq \mathsf{Cast}\langle \mathsf{d}' :: \mathsf{A}' \sqsubseteq \mathsf{A}; \mathsf{v}, \mathsf{v}' : \mathsf{A} \triangleleft \mathsf{B}\rangle)^{\mathsf{e}}]\!] = [\![(\mathsf{d}''' :: \mathsf{A}'' \sqsubseteq \mathsf{A}')^{\mathsf{e}}]\!]$, immediate by induction.
(5) Functorial rules (sum, product, function): each can be verified to implement the functorial action of the type connective, which as observed above necessarily preserves ep pairs.

□

$$\frac{}{\mathsf{Imp}(?) :: ? \sqsubseteq ?}\ \textsc{Id}$$

$$\frac{}{\mathsf{Imp}(\mathbb{N}) :: \mathbb{N} \sqsubseteq ?}\ \textsc{Tag}$$

$$\frac{}{\mathsf{Imp}(1) :: 1 \sqsubseteq ?}\ \textsc{Tag}$$

$$\frac{}{\mathsf{Imp}(0) :: 0 \sqsubseteq ?}\ \textsc{Empty}$$

$$\dfrac{\dfrac{\mathsf{Imp}(A) :: A \sqsubseteq ? \qquad \mathsf{Imp}(B) :: B \sqsubseteq ?}{A \times B \sqsubseteq ? \times ?}\ \textsc{Pair} \qquad \dfrac{}{? \times ? \sqsubseteq ?}\ \textsc{Tag}}{\mathsf{Imp}(A \times B) :: A \times B \sqsubseteq ?}\ \textsc{Cut}$$

$$\dfrac{\dfrac{\mathsf{Imp}(A) :: A \sqsubseteq ? \qquad \mathsf{Imp}(B) :: B \sqsubseteq ?}{A + B \sqsubseteq ? + ?}\ \textsc{Sum} \qquad \dfrac{}{? + ? \sqsubseteq ?}\ \textsc{Tag}}{\mathsf{Imp}(A + B) :: A + B \sqsubseteq ?}\ \textsc{Cut}$$

$$\dfrac{\dfrac{\mathsf{Imp}(A) :: A \sqsubseteq ? \qquad \mathsf{Imp}(B) :: B \sqsubseteq ?}{A \to B \sqsubseteq ? \to ?}\ \textsc{Fun} \qquad \dfrac{}{? \to ? \sqsubseteq ?}\ \textsc{Tag}}{\mathsf{Imp}(A \to B) :: A \to B \sqsubseteq ?}\ \textsc{Cut}$$

$$\mathsf{Imp}(\mathsf{Cast}\langle d :: A' \sqsubseteq A; v_e, v_p : A \triangleleft B\rangle) = \text{cut of the following 3 derivations :}$$

$$\frac{d :: A' \sqsubseteq A}{\mathsf{Cast}\langle d :: A' \sqsubseteq A; v_e, v_p : A \triangleleft B\rangle \sqsubseteq \mathsf{Cast}\langle v_e, v_p : A \triangleleft B\rangle}\ \textsc{Cast-Cong}$$

$$\frac{}{\mathsf{Cast}\langle v_e, v_p : A \triangleleft B\rangle \sqsubseteq B}\ \textsc{Cast-Tag}$$

$$\mathsf{Imp}(B) :: B \sqsubseteq ?$$

Fig. 28. Canonical Imprecision Judgment

Next, coherence

Theorem A.44 (Type Precicion Judgments are EP Pair factorizations). *For any derivation* $d :: A \sqsubseteq B$, $[\![A^e]\!] = [\![B^e]\!] \circ [\![d^e]\!]$ *and* $[\![A^p]\!] = [\![d^p]\!] \circ [\![B^p]\!]$

Proof. By induction on d. All cases are symmetric in embedding and projections, so we only detail the embedding case.

Case $\mathrm{id} :: A \sqsubseteq A$: trivial

Case d is the composition of $d_1 :: A \sqsubseteq C$, $d_2 :: C \sqsubseteq B$. Then we know by inductive hypothesis that $[\![B^e]\!] \circ [\![{d_2}^e]\!] = [\![C^e]\!]$ and $[\![C^e]\!] \circ [\![{d_1}^e]\!] = [\![A^e]\!]$ so we get $[\![A^e]\!] = [\![B^e]\!] \circ [\![{d_2}^e]\!] \circ [\![{d_1}^e]\!]$.

Case d is the tagging $G \sqsubseteq ?$. In this case $[\![B^e]\!] = [\![?^e]\!] = \mathrm{id}$, so we need to show $[\![G^e]\!] = [\![d^e]\!]$. If G is a base type $1, \mathbb{N}$ then this is true by definition.

Otherwise, G is a connective applied to the dynamic type in every position, $? \times ?, ? + ?, ? \to ?$ and $[\![G^e]\!]$ is the composition of the tagging rule with the congruence rule applied to the identity $? \sqsubseteq ?$ in ever position. Since the semantics of the identity rule is the identity and the semantics of the functorial rule is a functorial action, this is the composition of the tagging rule with the identity so we are done.

Case If $d :: \mathrm{Cast}\langle d'' :: A'' \sqsubseteq A; v, v' : A \triangleleft B\rangle \sqsubseteq \mathrm{Cast}\langle d' :: A' \sqsubseteq A; v, v' : A \triangleleft B\rangle$ is the cast congruence rule with some further subderivation $d_m :: A'' \sqsubseteq A'$ then our three inductive hypotheses are

$$[\![A''^e]\!] = [\![A^e]\!] \circ [\![d''^e]\!]$$
$$[\![A'^e]\!] = [\![A^e]\!] \circ [\![d'^e]\!]$$
$$[\![A''^e]\!] = [\![A'^e]\!] \circ [\![d_m{}^e]\!]$$

and inspecting the definition of $\mathrm{Imp}(\,)$ for a cast refinement, we see that it is sufficient to show

$$[\![d''^e]\!] = [\![d'^e]\!] \circ [\![d_m{}^e]\!]$$

By combining inductive hypotheses we get

$$[\![A^e]\!] \circ [\![d''^e]\!] = [\![A''^e]\!] = [\![A^e]\!] \circ [\![d'^e]\!] \circ [\![d_m{}^e]\!]$$

and the result follows because $[\![A^e]\!]$ is injective (i.e., mono) because it has a retraction.

Case If $d :: \mathrm{Cast}\langle \mathrm{id} :: A \sqsubseteq A; v, v' : A \triangleleft B\rangle \sqsubseteq B$ is the cast tagging rule, then the result holds by definition of $\mathrm{Imp}(\mathrm{Cast}\langle \mathrm{id} :: A \sqsubseteq A; v, v' : A \triangleleft B\rangle)$ and the fact that $[\![\mathrm{id}^e]\!] = \mathrm{id}$

Case If $d = d_1 \to d_2$ with $A = A_1 \to A_2, B = B_1 \to B_2$. Then

$$[\![A_1 \to A_2{}^e]\!] = [\![A_1{}^p]\!] \to [\![A_2{}^e]\!]$$
$$[\![B_1 \to B_2{}^e]\!] = [\![B_1{}^p]\!] \to [\![B_2{}^e]\!]$$
$$[\![d_1 \to d_2{}^e]\!] = [\![d_1{}^p]\!] \to [\![d_2{}^e]\!]$$

and we need to show that

$$[\![A_1{}^p]\!] \to [\![A_2{}^e]\!] = ([\![B_1{}^p]\!] \to [\![B_2{}^e]\!]) \circ ([\![d_1{}^p]\!] \to [\![d_2{}^e]\!])$$

by functoriality we have

$$([\![B_1{}^p]\!] \to [\![B_2{}^e]\!]) \circ ([\![d_1{}^p]\!] \to [\![d_2{}^e]\!]) = ([\![d_1{}^p]\!] \circ [\![B_1{}^p]\!]) \to ([\![B_2{}^e]\!] \circ [\![d_2{}^e]\!])$$

and the result holds by induction.

The sum, product case are directly analogous to the function case.

□

A.2.6 Semantic Term Precision. Now we prove the gradual guarantee by constructing a logical relation interpreting type precision semantically and proving soundness of term precision with respect to this semantics.

First, we define our universe of relations

Definition A.45 (Admissible Error Approximation Relations). (1) An (admissible error) approximation relation $R : X \nrightarrow Y$ on predomains X, Y is a subset $R \subset |X| \times |Y|$ that is down-closed in $\sqsubseteq_X$ and upper-closed in $\sqsubseteq_Y$, meaning if $x' \sqsubseteq_X x, y \sqsubseteq_Y y'$ and xRy then $x'Ry'$ and respects directed joins in $\leq_X, \leq_Y$ in that for any $D \subset X, E \subset Y$ $\leq_{\mathrm{div}}$-directed, if for every $x \in D, y \in E$ there exists $x' \in D, y' \in E$ with $x \leq_X x', y \leq_Y y'$ and $x'Ry'$ then $(\bigvee D)\ R\ (\bigvee E)$.

(2) An (admissible error) approximation relation $R : X \twoheadrightarrow Y$ on error domains X, Y is a subset $R \subset |X| \times |Y|$ that is an admissible error approximation on $U_{\mho}X, U_{\mho}Y$ with $\mho_X R \mho_Y$.

(3) For any $R : X \twoheadrightarrow Y$ approximation relation on predomains, there is an approximation relation $R_{\Omega,\mho} : U_{\Omega}F_{\Omega,\mho}X \twoheadrightarrow U_{\Omega}F_{\Omega,\mho}Y$ defined by:

$$\mho_X \; R_{\Omega,\mho} \; y$$
$$\Omega_X \; R_{\Omega,\mho} \; \Omega_Y$$
$$\lfloor x \rfloor \; R_{\Omega,\mho} \; \lfloor y \rfloor = xRy$$

Definition A.46 (Preservation of Relations). If $f : X \to_{\Omega,\mho} Y, g : X' \to_{\Omega,\mho} Y$ and $R : X \twoheadrightarrow X', S : Y \twoheadrightarrow Y'$, we say (f, g) takes R to S, denoted $f \; R \longrightarrow S \; g$ if for any $x \; R \; y$, $f(x) \; S_{\Omega,\mho} \; g(y)$.

Then we can again reframe inclusion of relations as preservation by identities:

Lemma A.47 (Inclusion is Identity Preservation). (1) *If $R, S : X \twoheadrightarrow Y$, then $R \subseteq S$ if and only if $id_X \; R \longrightarrow S \; id_Y$.*

Composition of approximation relations is not, in general, admissible,

Definition A.48 (Composition of Relations). When it is admissible, we define the composite of $R : X \twoheadrightarrow Y, S : Y \twoheadrightarrow Z$ to be the relation:

$$x \; R; S \; z \overset{\text{def}}{=} \exists y. x \; R \; y \wedge y \; S \; z$$

We will show later that all of the compositions we need are well-defined. For now the following it is useful to note that when defined, composition is associative, and the error orderings $\sqsubseteq_{\text{err}}$ act as the identity for composition:

Lemma A.49 (Identity, Associativity of Relational Composition). (1) *For any $R : X \twoheadrightarrow Y$, $R; \sqsubseteq_Y = R = \sqsubseteq_X; R$.*

(2) *For any $R : W \twoheadrightarrow X, S : X \twoheadrightarrow Y, T : Y \twoheadrightarrow Z$, if all composites are defined then $R; (S; T) = (R; S); T$.*

Relations and related functions form a category, giving us useful reasoning principles for proving preservation of relations. Furthermore, they *almost* form a double category in that we get composition along composite relations when the composites are well defined.

Lemma A.50 (Identity, horizontal and vertical Composition of Preservation). (1) *For any $R : X \twoheadrightarrow Y$, $id_X \; R \longrightarrow R \; id_Y$.*

(2) *For any $R_1 : X_1 \twoheadrightarrow Y_1, R_2 : X_2 \twoheadrightarrow Y_2, R_3 : X_3 \twoheadrightarrow Y_3$, if $f \; R_1 \longrightarrow R_2 \; g$ and $h \; R_2 \longrightarrow R_3 \; k$, then $hf \; R_1 \longrightarrow R_3 \; kg$. We call this* vertical *composition.*

(3) *For any $R_1 : X_1 \twoheadrightarrow Y_1, S_1 : Y_1 \twoheadrightarrow Z_1, R_2 : X_2 \twoheadrightarrow Y_2, S_2 : Y_2 \twoheadrightarrow Z_2$ if $R_1; S_1$ and $R_2; S_2$ are admissible, and $f \; R_1 \longrightarrow R_2 \; g$ and $g \; S_1 \longrightarrow S_2 \; h$ then $f \; (R_1; S_1) \longrightarrow (R_2; S_2) \; h$. We call this* horizontal *composition*

Proof. (1) equivalent to $R \subseteq R$.

(2) Let $x_1 \; R_1 \; y_1$. Then if $f(x_1) = \mho, h(f(x_1)) = \mho$ and we're done. Similarly if $f(x_1) = \Omega, g(y_1) = \Omega$ so $h(f(x_1)) = \Omega, k(g(y_1)) = \Omega$. Finally if $f(x_1) = \lfloor x_2 \rfloor$ then $g(y_1) = \lfloor y_2 \rfloor$ and the result follows by relatedness of h, k.

(3) Let $x_1 \; R_1 \; y_1 \; S_1 \; z_1$, then if $f(x_1) = \mho$ we're done. If $f(x_1) = \Omega$, then $g(y_1) = \Omega$ so $h(z_1) = \Omega$. If $f(x_1) = \lfloor x_2 \rfloor$, then $g(y_1) = \lfloor y_2 \rfloor$ with $x_2 \; R_2 \; y_2$ and $h(z_1) = \lfloor z_2 \rfloor$ with $y_2 \; S_2 \; z_2$.

□

Next, we can build approximation relations from ep pairs, or more generally we can take the "pullback" of any pure function and "pushforward" of any total function, i.e., one that may error but does not diverge.

Definition A.51 (Pullback and Pushforward). (1) If $f : X \to Y$ is a continuous function of predomains, then the pullback of f, denoted $f^* : X \mathrel{\ooalign{$\to$\cr\hidewidth$+$\hidewidth}} Y$ defined by

$$x\ f^*\ y = f(x) \sqsubseteq_Y y$$

is an admissible relation of predomains.

(2) If $g : Y \to F_{\mho}X$ is a continuous erroring function of predomains, then the pushforward of g, denoted $g_! : X \mathrel{\ooalign{$\to$\cr\hidewidth$+$\hidewidth}} Y$ defined by

$$x\ g_!\ y \overset{\text{def}}{=} \lfloor x \rfloor \sqsubseteq_{F_{\mho}X} g(y)$$

is an admissible relation of predomains.

Crucially, composition with a pullback on the left or pushforward on the right is *always* admissible.

Lemma A.52 (Composition with pullback, pushforward). (1) *If $R : X \mathrel{\ooalign{$\to$\cr\hidewidth$+$\hidewidth}} Y$ and $f : W \to X$, then $f^*; R : W \mathrel{\ooalign{$\to$\cr\hidewidth$+$\hidewidth}} Y$ is admissible*

(2) *If $R : X \mathrel{\ooalign{$\to$\cr\hidewidth$+$\hidewidth}} Y$ and $g : Z \to F_{\mho}Y$, then $R; g_! : X \mathrel{\ooalign{$\to$\cr\hidewidth$+$\hidewidth}} Z$ is admissible.*

Proof. Easy after noting that $w\ (f^*; R)\ y$ if and only if $f(w)Ry$ and $x\ (R; g_!)\ z$ if and only if $g(z) = \lfloor y \rfloor$ and $x\ R\ y$. □

The following key technical lemma (adapted from [23]) is what enables our proof that semantic term precision has the structure of a logical relation. It says that the pushforward and pullback are uniquely determined by two properties that we will show are preserved by all type constructors:

Lemma A.53 (Universal Property of pullback, pushforward). (1) *For any $f : X \to Y$ the pullback f^* is the unique relation satisfying $f\ (f^* \to \sqsubseteq_Y)\ id_Y$ and $id_X\ (\sqsubseteq_X \to f^*)\ f$.*

(2) *For any $g : Y \to F_{\mho}X$, the pushforward $g_! : X \mathrel{\ooalign{$\to$\cr\hidewidth$+$\hidewidth}} Y$ is the unique relation satisfying $id_X\ (g_! \to \sqsubseteq_X)\ g$ and $g \sqsubseteq_Y \to g_!\ id_Y$.*

Proof. That they satisfy the properties given is an easy calculation. To see that f^* is unique, given any other R satisfying the same properties we have $\mathrm{id}_X\ (\sqsubseteq_X \to f^*)\ f$ and $f\ (R \to \sqsubseteq_Y)\ \mathrm{id}_Y$ so by composition we have

$$\mathrm{id}_X \sqsubseteq_X; R \to f^*; \sqsubseteq_Y$$

so $R \subseteq f^*$. A symmetric argument shows the other direction and the same for $g_!$. □

Finally, we can extend the universal property above to arbitrary relations, not necessarily identities:

Lemma A.54 (Pullback, pushforward reasoning principle). *Whenever the pullbacks, pushforwards exist,*

(1) *$f\ R \to S\ g \circ h$ if and only if $f\ R; h^* \to S\ g$*
(2) *$f\ R \to S\ g \circ h$ if and only if $f\ R \to S; g_!\ h$*
(3) *$f \circ k\ R \to S\ g$ if and only if $f\ k_! R \to S\ g$*
(4) *$f \circ k\ R \to S\ g$ if and only if $k\ R \to f^* S\ g$*

Now we will construct an appropriate notion of action of a functor on relations:

Definition A.55 (Admissible Action of a Functor on Relations). Given a functor $F : (\mathrm{PreDom}_{\mho}^{op})^n \times \mathrm{PreDom}_{\mho}^m \to \mathrm{PreDom}_{\mho}$, an admissible action of F on relations is a mapping:

$F : (X_1 \nrightarrow X_1')^{op} \times \cdots \times (X_n \nrightarrow X_n')^{op} \times (Y_1 \nrightarrow Y_1') \times \cdots \times (Y_m \nrightarrow Y_m) \to (F(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \nrightarrow F(X_1', \ldots, X_n'$

satisfying:

(1) Functoriality: If $f_1\ P_1 \to R_1\ f_1', \ldots, g_1\ S_1 \to T_1\ g_1', \ldots$ then

$$F(f_1, \ldots, g_1, \ldots)\ F(R_1, \ldots, S_1, \ldots) \to F(P_1, \ldots, T_1, \ldots)\ F(f_1', \ldots, g_1', \ldots)$$

(2) Normality/Identity Extension: $F(\sqsubseteq_{X_1}, \ldots, \sqsubseteq_{X_m}, \sqsubseteq_{Y_1}, \ldots, \sqsubseteq_{Y_m}) = \sqsubseteq_{F(X_1, \ldots, X_n, Y_1, \ldots, Y_m)}$.

(3) Effect Neutrality: If all the g_i are pure and f_i are total, then $F(f_1, \ldots, g_1, \ldots)$ is pure and if all the f_i, g_i are total, $F(f_1, \ldots, g_1, \ldots)$ is total.

We will call such an F a relational functor.

Now we can use our technical lemma to prove that all constructors we use in type precision commute with pullback, pushforward.

Lemma A.56 (Everything Preserves Pullback, Pushforward). (1) $(g \circ f)^* = f^*; g^*$

(2) $(g \circ f)_! = g_!; f_!$

(3) $id_X^* = \sqsubseteq_X$

(4) $id_{X!} = \sqsubseteq_X$

(5) *For any relational functor F,*

$$F(f_{1!}, \ldots, f_{n!}, g_1^*, \ldots, g_m^*) = F(f_1, \ldots, f_n, g_1, \ldots, g_m)^*$$

(6) *For any relational functor F,*

$$F(f_1^*, \ldots, f_n^*, g_{1!}, \ldots, g_{1!}) = F(f_1, \ldots, f_n, g_1, \ldots, g_m)_!$$

Proof. We consider the pullback cases, the pushforwards are dual.

(1) Suffices to show $g \circ f\ f^*; g^* \to \sqsubseteq_Z \mathrm{id}_Z$. First, $f\ f^*; g^* \to g^*\ \mathrm{id}_Z$ by horizontal composition of $f\ f^* \to \sqsubseteq_Y \mathrm{id}_Y$ and $\mathrm{id}_Y\ g^* \to g^*\ \mathrm{id}_Z$ and $\sqsubseteq_Y; g^* = g^*$. Then we get the result by vertical composition with $g\ g^* \to \sqsubseteq_Z \mathrm{id}_Z$. The other case is dual.

(2) Dual

(3) Obvious

(4) Obvious

(5) For clarity, say $f_i : W_i \to_{\mho} X_i, g_j : Y \to Z$ By effect neutrality, the pullback of $F(f, \ldots, g, \ldots)$ is well defined.

Since $f_i \sqsubseteq_{W_i} \to f_{i!}\ \mathrm{id}_{W_i}, g_j\ g_j^* \to \sqsubseteq_{Z_j} \mathrm{id}_{Z_j}$ by functoriality we have

$$F(f_{1!}, \ldots, g_1^*)\ F(f_{1!}, \ldots, g_1^*) \to F(\sqsubseteq_{W_1}, \ldots, \sqsubseteq_{Z_1}, \ldots)\ F(\mathrm{id}, \ldots)$$

But since F is a functor, $F(\mathrm{id}, \ldots) = \mathrm{id}$ and by normality, $F(\sqsubseteq_{W_1}, \ldots, \sqsubseteq_{Z_1}, \ldots) = \sqsubseteq_{F(\ldots)}$. The other case is dual.

(6) Dual to the previous.

□

And to transport our factorization proofs on ep pairs to those on relations, we observe that taking pullbacks/pushforwards is faithful on ordering:

Lemma A.57. *Pullbacks, Pushforwards are faithful*

(1) *$f \sqsubseteq_{X \to Y} g$ if and only if $g^* \subseteq f^*$.*

(2) *$f \sqsubseteq_{X \to_{\mho} Y} g$ if and only if $f_! \subseteq g_!$.*

Proof. If $f \sqsubseteq_{X \to Y} g$, then if $g(x) \sqsubseteq_Y y$, $f(x) \sqsubseteq_Y g(x) \sqsubseteq_Y y$ by transitivity. If $g^* \subseteq f^*$, then for any x, $x\ g^*\ g(x)$ so $x\ f^*\ g(x)$, which is just $f(x) \sqsubseteq_Y g(x)$. The pushforward case is dual. □

Corollary A.58 (Coherence for Relational Semantics). *If $d, d' :: \mathsf{A} \sqsubseteq \mathsf{A}'$, then $\sqsubseteq_d = \sqsubseteq_{d'}$.*

So to prove our relational semantics gives us the intended semantics of the gradual guarantee we verify that all of our type connectives extend to relational functors

Lemma A.59 (Type Constructors are Relational Functors). *All the relational actions of functors defined in Figure 29 make the denotations of type constructors relational functors.*

Theorem A.60 (Relational and EP-pair Semantics Agree). *For any well formed $d :: \mathsf{A} \sqsubseteq \mathsf{A}'$,*

$$\sqsubseteq_d = [\![d^{\mathbf{e}}]\!]^* = [\![p^{\mathbf{p}}]\!]_!$$

Proof. By induction on precision derivations, using Lemma ??. □

The final proof of the fundamental lemma is straightforward since it follows the structure of a usual logical relation, and we can isolate the reasoning about type precision using the following lemma:

Lemma A.61 (Cast Compatibility). (1) *If $\mathsf{A} \sqsubseteq \mathsf{A}'$ and $\mathsf{B} \sqsubseteq \mathsf{B}'$, then*

$$[\![(\mathsf{A} \implies \mathsf{B})^{+}]\!] \sqsubseteq_{\mathsf{A} \sqsubseteq \mathsf{A}'} \to \sqsubseteq_{\mathsf{B} \sqsubseteq \mathsf{B}'} [\![(\mathsf{A}' \implies \mathsf{B}')^{+}]\!]$$

(2) *If $\mathsf{A} \sqsubseteq \mathsf{A}'$ then*

$$[\![\mathsf{A}^{\mathbf{e}}]\!] \sqsubseteq_{\mathsf{A} \sqsubseteq \mathsf{A}'} \to \sqsubseteq_{[\![?]\!]} [\![\mathsf{A}'^{\mathbf{e}}]\!]$$

(3) *If $\mathsf{B} \sqsubseteq \mathsf{B}'$ then*

$$[\![\mathsf{A}^{\mathbf{p}}]\!] \sqsubseteq_{[\![?]\!]} \to \sqsubseteq_{\mathsf{B} \sqsubseteq \mathsf{B}'} [\![\mathsf{A}'^{\mathbf{p}}]\!]$$

Proof. (1) Follows from the next 2 by composition

(2) By Lemma A.54 and Lemma A.50, we equivalently show

$$\sqsubseteq_{\mathsf{A} \sqsubseteq \mathsf{A}'}; [\![\mathsf{A}'^{\mathbf{e}}]\!]^* \subset [\![\mathsf{A}^{\mathbf{e}}]\!]^*; \sqsubseteq_{[\![?]\!]}$$

By Theorem A.60, $[\![\mathsf{A}'^{\mathbf{e}}]\!]^* = \sqsubseteq_{\mathsf{A}' \sqsubseteq ?}$ and $\sqsubseteq_{\mathsf{A} \sqsubseteq \mathsf{A}'}; [\![\mathsf{A}'^{\mathbf{e}}]\!]^* = \sqsubseteq_{\mathsf{A} \sqsubseteq ?}$, and similarly for the right hand side $[\![\mathsf{A}^{\mathbf{e}}]\!]^*; \sqsubseteq_{[\![?]\!]} = \sqsubseteq_{\mathsf{A} \sqsubseteq ?}$, so the two relations are in fact equal.

(3) Dual to previous.

□

Lemma A.62 (Fundamental Lemma for Term Precision Soundness). *If*

$$\Gamma \vdash \mathsf{t} : \mathsf{A} \sqsubseteq \Gamma' \vdash \mathsf{t}' : \mathsf{A}'$$

then

$$[\![\mathsf{t}]\!]\ (\sqsubseteq_{\Gamma \sqsubseteq \Gamma'} \to \sqsubseteq_{\mathsf{A} \sqsubseteq \mathsf{A}'})\ [\![\mathsf{t}']\!]$$

Proof. By induction on term precision derivations. All cases involving casts use Lemma A.61. Again we go through the recursion case in detail:

Given

$$\Gamma \vdash \mathsf{rec}\,\mathsf{x}\,\mathsf{y}\,.\,\mathsf{t} : \mathsf{A} \sqsubseteq \Gamma' \vdash \mathsf{rec}\,\mathsf{x}\,\mathsf{y}\,.\,\mathsf{t}' : \mathsf{A}'$$

, by inductive hypothesis we know

$$[\![\mathsf{t}]\!]\ (\sqsubseteq_{\Gamma \sqsubseteq \Gamma'} \times \sqsubseteq_{\mathsf{A} \to \mathsf{B} \sqsubseteq \mathsf{A} \to \mathsf{B}} \times \sqsubseteq_{\mathsf{A} \sqsubseteq \mathsf{B}} \to \sqsubseteq_{\mathsf{B} \sqsubseteq \mathsf{B}'})\ [\![\mathsf{t}']\!]$$

and we want to show that

$$
\begin{array}{rcl}
; & : & (X \nrightarrow Y) \times (Y \nrightarrow Z) \to (X \nrightarrow Z)\\
x\; R;S\; z & \overset{\text{def}}{=} & \exists y. x\; R\; y \wedge y\; S\; z\\
\sqsubseteq_X & : & (X \nrightarrow X)\\
0_X & : & (0 \nrightarrow X)\\
d\; 0_X\; x & \overset{\text{def}}{=} & \bot\\
\to & : & (X \nrightarrow X')^{op} \times (Y \nrightarrow Y') \to ((X \to U_\Omega F_{\Omega,\mho} Y) \nrightarrow (X' \to U_\Omega F_{\Omega,\mho} Y'))\\
f\; R \to S\; g & \overset{\text{def}}{=} & \forall x\; R\; y. f(x)\; S_{\Omega,\mho}\; g(y)\\
+ & : & (X \nrightarrow X') \times (Y \nrightarrow Y') \to ((X + Y) \nrightarrow (X' + Y'))\\
\sigma_i x\; R_1 + R_2\; \sigma_j y & \overset{\text{def}}{=} & i = j \wedge x\; R_i\; y\\
\Pi_{i \in I} & : & (X_{i_1} \nrightarrow X'_{i_1}) \times \cdots \times (X_{i_{|I|}} \nrightarrow X'_{i_{|I|}}) \to \Pi_{i\in I} X_i \nrightarrow \Pi_{i \in I} X'_i\\
x\; (\Pi_{i\in I} R_i) x' & \overset{\text{def}}{=} & \forall i \in I. \pi_i x\; R_i\; \pi_i x'
\end{array}
$$

Fig. 29. Constructions on Relations

$$\lambda\gamma.Y(\lambda f.\lambda x.[\![t]\!](\gamma, f, x))\; (\sqsubseteq_{\Gamma\sqsubseteq\Gamma'} \to \sqsubseteq_{B\sqsubseteq B'})\; \lambda\gamma'.Y(\lambda f'.\lambda x'.[\![t']\!](\gamma', f', x'))$$

By admissibility, it is sufficient to show that for any $n \in \omega$ and $\gamma \sqsubseteq_{\Gamma\sqsubseteq\Gamma'} \gamma'$,

$$g^n(\lambda y.\Omega) \sqsubseteq_{B\sqsubseteq B'} g'^n(\lambda y'.\Omega)$$

where $g = \lambda f.\lambda x.[\![t]\!](\gamma, f, x), g' = \lambda f'.\lambda x'.[\![t']\!](\gamma', f', x')$ which follows by an easy induction: If $n = 0$, we need

$$\lambda y.\Omega(\sqsubseteq_{A\sqsubseteq A'} \to \sqsubseteq_{B\sqsubseteq B'})\lambda y.\Omega$$

which is immediate from the definition of $\to$ on relations, and for $n+1$ given any $f\; (\sqsubseteq_{A\sqsubseteq A'} \to \sqsubseteq_{B\sqsubseteq B'})\; f'$, we have

$$g(f)\; (\sqsubseteq_{A\sqsubseteq A'} \to \sqsubseteq_{B\sqsubseteq B'})\; g'(f')$$

i.e.,

$$\lambda x.[\![t]\!](\gamma, f, x)\; (\sqsubseteq_{A\sqsubseteq A'} \to \sqsubseteq_{B\sqsubseteq B'})\; \lambda x'.[\![t']\!](\gamma', f', x')$$

which follows from our inductive hypothesis about t, t'. □

Theorem A.63 (Soundness of Term Precision: Gradual Guarantee). *If $\Gamma \vdash t : A \sqsubseteq \Gamma' \vdash t' : A'$ then $t \sqsubseteq^{hctx}_{err} t'$*

Proof. Given $C : (\Gamma'' \vdash B) \Rightarrow (\cdot \vdash 1)$ where $\Gamma, \Gamma', \Gamma''$ use exactly the same variables $x_1, ldots, x_n$, we need to show that

$$C\langle t\rangle \sqsubseteq_{\text{err}} C\langle t'\rangle$$

By Theorem A.19, it is sufficient to show

$$[\![C\langle t\rangle]\!] \sqsubseteq_{1_{\Omega,\mho}} [\![C\langle t'\rangle]\!]$$

Let $\Gamma_? = x_1 : ?, \ldots, x_n : ?$. Then by Lemma A.17,

$$[\![C\langle t\rangle]\!] = [\![C]\!]([\![(A \implies B)^+]\!] \circ [\![t]\!] \circ [\![(\Gamma'' \implies \Gamma)^+]\!])$$
$$[\![C\langle t'\rangle]\!] = [\![C]\!]([\![(A' \implies B)^+]\!] \circ [\![t]\!] \circ [\![(\Gamma'' \implies \Gamma')^+]\!])$$

So by monotonicity of application it is sufficient to show

$$[\![(A \implies B)^+]\!] \circ [\![t]\!] \circ [\![(\Gamma'' \implies \Gamma)^+]\!] \sqsubseteq_{[\![\Gamma'']\!] \to [\![B]\!]_{\Omega,\mho}}$$

Which follows by compositionality, Lemma A.62 and Lemma A.61. □

$$
\begin{array}{rcl}
\sqsubseteq_{\mathrm{id}::\mathsf{A}\sqsubseteq\mathsf{A}} & \overset{\text{def}}{=} & \sqsubseteq_{[\![\mathsf{A}]\!]} \\
\sqsubseteq_{d_1;d_2::\mathsf{A}\sqsubseteq\mathsf{C}} & \overset{\text{def}}{=} & \sqsubseteq_{d_1} ; \sqsubseteq_{d_2} \\
\sqsubseteq_{\mathrm{Tag}(\mathsf{G})::\mathsf{G}\sqsubseteq ?} & \overset{\text{def}}{=} & [\![(\mathrm{Tag}(\mathsf{G}))^{\mathbf{e}}]\!]^{*} \\
\sqsubseteq_{0::0\sqsubseteq\mathsf{A}} & \overset{\text{def}}{=} & 0_{[\![\mathsf{A}]\!]} \\
\sqsubseteq_{d_1+d_2::\mathsf{A}\sqsubseteq\mathsf{C}} & \overset{\text{def}}{=} & \sqsubseteq_{d_1} + \sqsubseteq_{d_2} \\
\sqsubseteq_{d_1\times d_2::\mathsf{A}\sqsubseteq\mathsf{C}} & \overset{\text{def}}{=} & \sqsubseteq_{d_1} \times \sqsubseteq_{d_2} \\
\sqsubseteq_{d_1\to d_2::\mathsf{A}\sqsubseteq\mathsf{C}} & \overset{\text{def}}{=} & \sqsubseteq_{d_1} \to \sqsubseteq_{d_2} \\
\sqsubseteq_{\mathrm{CastTag}::\mathsf{Cast}\langle \mathsf{v}_e,\mathsf{v}_p:\mathsf{A}\triangleleft\mathsf{B}\rangle\sqsubseteq\mathsf{B}} & \overset{\text{def}}{=} & [\![\mathsf{v}_e]\!]^{*} \\
\sqsubseteq_{\mathrm{CastCong}(d)::\mathsf{Cast}\langle \mathsf{A}''\sqsubseteq\mathsf{A};\mathsf{v},\mathsf{v}':\mathsf{A}\triangleleft\mathsf{B}\rangle\sqsubseteq\mathsf{Cast}\langle \mathsf{A}'\sqsubseteq\mathsf{A};\mathsf{v},\mathsf{v}':\mathsf{A}\triangleleft\mathsf{B}\rangle} & \overset{\text{def}}{=} & \sqsubseteq_{d::\mathsf{A}''\sqsubseteq\mathsf{A}'}
\end{array}
$$

Fig. 30. Relational Semantics of Type Precision