A Semantic Foundation for Sound Gradual Typing

Max S. New

October 7, 2019

1 Thesis

1.1 Problem

Gradually typed languages allow for statically typed and dynamically typed programs to interoperate with minimal programming overhead, and to support the gradual migration from a dynamically typed to a statically typed programming discipline. Gradual migration allows for dynamically typed prototypes to be hardened with static types over time, providing machine-checked documentation, static detection of errors when writing extensions and improved compiler optimizations. In addition, static-dynamic interoperability allows for easy use of long-lived typed libraries by ephemeral, dynamically typed clients without imposing static discipline on one-off scripts, providing a kind of safe "FFI" between static and dynamic fragments of the language.

In addition to the desirability of these as programming features, there are also sociological reasons for the development of extensions of existing dynamic languages to incorporate gradual typing. While type theoretic principles have been immensely successful in the semantic analysis of programming features, there is a vast amount of code already written in dynamically typed languages, and gradual migration to stricter typing disciplines may be preferable to wholesale rewrites to a new language.

Since the introduction of gradual typing, first codified in two independent works [Tobin-Hochstadt and Felleisen, 2008, Siek and Taha, 2006], the area has seen a great proliferation of work designing gradual calculi for various typing features, but also on the practical side of adapting existing dynamic languages and developing more efficient implementations.

However, the semantic study of gradual languages has severely lagged behind the vast number of languages and calculi that have been developed. First, despite the fact that the purpose of introducing typing is to have machine-checked, actionable type information embedded in programs, very little prior work has proven that gradual typing disciplines ensure the safety and correctness of compiler optimizations or program transformations. Typical theorems, such as "blame soundness" are quite weak and some cast calculi provide very weak guarantees despite satisfying theorems that they call "type soundness".

One semantic aspect of gradual typing that has seen some recent research is the *graduality* theorem, originally named the "gradual guarantee" [Siek et al., 2015]. The graduality theorem ensures that the dynamic type checking introduced by changing the precision of types never "gets in the way" of the programmer. Intuitively it says that if the types in a program are made more precise then if the program still type checks, either the program has the same dynamic behavior as the original a dynamic type error. This allows for a separation of concerns for programmers: they know that if the original program's behavior was correct, then adding types will not change this.

Finally, most gradual languages are designed in an ad hoc manner, with certain soundness theorems as goals, but no systematic theory of how to design languages that both justify type-based optimizations and satisfy graduality. My dissertation seeks to address this by developing a new semantic theory of gradual typing that helps to develop novel gradual languages that satisfy these strong soundness theorems.

1.2 Thesis

> The theory of embedding-projection pairs provides a common semantic framework for the design and metatheory of sound gradually typed languages.

The meaning of *sound gradually typed language* is controversial, with different languages claiming a wide variety of soundness theorems. For the purposes of this proposal, I mean languages that satisfy relational soundness criteria I present in Section 3, which combine appropriate β, η equations with a relational formulation of graduality. I say that the theory is *semantic* because the property that casts

are produced from embedding-projection pairs is entirely based on the behavior of the casts, and makes no appeal to a particular syntactic presentation of the types or terms of the surface gradual language. By a *common framework for design and metatheory*, I mean that the theory can be adapted with little change to a variety of different gradual languages, both for the purposes of informing the design of new languages and the proofs of our desired metatheoretic properties of relational type soundness and graduality.

The theory is based on a semantic analysis of the runtime type-checking of existing gradually typed languages; specifically that their behavior can be explained in terms of the semantic notion of an *embedding-projection pair* (sometimes abbreviated as ep pair). An embedding projection pair from a type A to a type B consists of a pair of functions called the *embedding* $e : A \to B$ and *projection* $p : B \to A$ which one can think of as casts between the types A and B where A is a more precise type and B is a laxer type, usually written $A \sqsubseteq B$. The pair e, p form an embedding-projection pair when they satisfy the following two properties:

1. (Embedding) for all $t : A$, $p(e(t)) \equiv t$. So every element of the more precise type can be safely cast to the laxer type and back.

2. (Projection) for all $u : B$, $e(p(u)) \sqsubseteq^{\mathtt{ctx}} u$, intuitively meaning either $e(p(u))$ has the same behavior as u or results in an error.

In Section 4, I show that proving certain casts in a gradual language form embedding-projection pairs leads to proofs of both correctness of optimizations and graduality.

Next, to demonstrate the generality of the embedding-projection pair semantics, I show in Section 5 that the theory can be applied to multiple evaluation orders (call-by-value, call-by-name and lazy evaluation). In fact, for simple types I strengthen the connection between sound gradual typing and embedding-projection pairs by showing that almost all of the cast semantics for these evaluation orders is entirely determined by the soundness theorems I have described. This reproduces various cast semantics previously given in the literature using the single unifying theory of embedding-projection pair semantics.

Finally, to show that embedding-projection pair semantics aids in the design of new calculi, in Section 6 I present a new calculus satisfying both data abstraction theorems for polymorphic types and graduality. This combination has been problematic in the literature and recent work has claimed the theorems to be incompatible [Igarashi et al., 2017, Ahmed et al., 2017, Toro et al., 2019]. I give a semantic model that generalizes the model from Section 4, and avoid the issues of previous work by departing from the syntax of System F and developing a novel syntax for gradual polymorphism based directly on our semantics. By focusing on the semantics first, I ensure that both parametricity theorems and graduality theorems hold by construction.

2 Prior Work

2.1 Retractions and Embedding-Projection Pairs

Our semantic model of gradual types as embedding-projection pairs into the dynamic type is analogous to Dana Scott's seminal work on the denotational semantics of programming languages with general recursion. However, I use a different ordering: Scott uses the domain ordering, which has an undefined element $\bot$ as bottom, which is used to interpret diverging programs, whereas our ordering has a runtime type error as bottom, in line with the graduality property.

2.2 Graduality and the Semantics of Contracts

The property I call *graduality* was originally named the *dynamic gradual guarantee* in Siek et al. [2015], but the idea that runtime type checking or contract checking should result in a refinement of program behavior has precedents in older works. Specifically, after Findler and Felleisen [2002] introduced contracts for higher-order functions, there were several proposals for semantic criteria for contracts. Most relevant for our purposes is the notion of contracts as *error projections* and the refinement to *pairs of projections*. While simple assertion-based contracts are presented by boolean predicates, higher-order contracts instead *coerce* values to behave according to the specification [Findler and Blume, 2006].

An error projection c on a type B is a function from B to itself satisfying two properties:

1. Idempotence: $c \circ c = c$

2. Projection: $c \sqsubseteq^{\mathtt{err}} \mathtt{id}$

The intuition is that c "forces" a value of B to behave according to some specification. Idempotence says that once something is forced to satisfy the specification, forcing it again makes no observable difference because it already satisfies the specification. Projection is analogous to graduality: it says that enforcing a contract only adds errors to a value's behavior. Here $\sqsubseteq^{\mathtt{err}}$ is not formally defined but is analogous to the ordering $\sqsubseteq^{\mathtt{ctx}}$ I will define in Definition 2.

There is a close technical connection between error projections and embedding-projection pairs. First, from any embedding-projection pair from A to B I can construct an error projection on B. Given an embedding $e : A \to B$ and corresponding projection $p : B \to A$, the composite $e \circ p : B \to B$ is an error projection. Idempotence follows from the retraction property, and projection from the projection property of ep pairs. In a sufficiently rich semantic setting, any error projection c can be "split" into an embedding-projection pair from B_c to B, where the elements of the domain B_c are defined to be the elements of B that are invariant under the projection: $c(b) = b$. Intuitively these elements are the ones that "satisfy" the contract: invariance means that applying the contract results in no observable change in behavior. This constitutes an equivalence between the concepts of an ep pair and an error projection, so our work can be seen as a continuation of this work that emphasizes a more intrinsically typed view of contracts. Furthermore, Findler and Blume [2006] models contracts as *pairs* of projections, one of which enforces the positive aspect of the contract and the other the negative. I argue that this idea is better modeled by embedding-projection pairs: the embedding enforces the negative aspects and the projection the positive aspects, but similar to "manifest" contracts [Greenberg et al., 2010] this is encoded in the type system so the embedding and projection are typed and mediate between a more precise and less precise type.

2.3 Type Consistency

The type systems of gradual languages in the style of Siek and Taha [2006] are based on notions of *consistency* and its refinement to *consistent subtyping*, exemplified by the following gradual typing rule for functions:

$$\frac{t : A \to B \qquad u : A' \qquad A' \lesssim A}{t\,u : B}$$

The consistent subtyping judgment $A \lesssim B$ is an extension of subtyping where ? is both a consistent subtype and a consistent supertype of every other type. This consistent subtyping judgment is what makes the type checking lax in the presence of the dynamic type: when A and A' are precise types, they must be actual subtypes, but if either contains ?, then type checking is less strict.

I will not provide a semantic explanation for these judgments in my dissertation, as it is not clear that they have a satisfying semantic explanation other than that they attempt to rule out "obvious" dynamic runtime errors. However, these have been related to precision in various works, especially the AGT framework [Garcia et al., 2016]. Specifically, one can view consistent subtyping $\lesssim$ as the minimal extension of subtyping $\leq$ that is *upper-closed* on both sides with respect to precision $\sqsubseteq$, i.e. the following rules are satisfied:

$$\frac{A \leq B}{A \lesssim B} \qquad\qquad \frac{A \sqsubseteq A' \qquad A \lesssim B \qquad B \sqsubseteq B'}{A' \lesssim B'}$$

This formalizes in a general way the notion that gradual subtyping is the result of "forcing" subtyping to be lax with respect to the dynamic type.

2.4 Frameworks for Gradual Typing

Some recently introduced frameworks for designing gradual languages, are similar in scope to our work, but I argue they fall short of satisfying my goals. The gradualizer framework [Cimini and Siek, 2016, 2017] is based on a system of heuristics out of which it is difficult to derive a general theory. The AGT framework [Garcia et al., 2016] on the other hand does provide a theory of gradual typing based on a semantics of gradual types as abstract interpretations of sets of static types, however it is not sufficiently flexible for our goals. The main drawback of these works for the purposes of this thesis is that they are quite rigid, providing a single language design for a gradual langauge that "gradualizes" a specific *statically typed* input language. As a consequence they cannot be used to adapt existing *dynamic* languages into gradual languages, in the style of Typed Racket, whereas the semantics of Typed

Racket can to a large extent be explained using embedding-projection pairs. Additionally, sometimes these frameworks produces languages that have semantic problems, and neither framework provides much aid in how to adapt the language design to compensate. For instance, the AGT framework only accounts for *eager* contract semantics of functions [Herman et al., 2010], which breaks certain η equivalences for functions. Furthermore, both frameworks, when applied to a language supporting parametric polymorphism, introduce violations of parametricity in the gradual language they produce. In both of these cases, the framworks do not provide guidance for how to change the language semantics to maintain the desired semantic principles, and at this point language designers revert back to ad hoc changes [Toro et al., 2018, 2019]. The embedding-projection pair semantics, on the other hand, provides a flexible *semantic* theory that can be adapted to different situations.

3 Relational Soundness Theorems for Gradual Typing

Before considering the semantics of gradual casts, let us consider what the desirable properties for a gradual language are. My key innovation here is to describe them in *relational* terms, which makes them amenable to semantic techniques such as logical relations (Section 4) and inequational theories (Section 5). Our approach is summarized by the slogan that a gradually typed language should be both *typed* and *gradual*, i.e., that it satisfies theorems saying that typing is sound and gradual changes to types cause simple behavioral changes.

3.1 Relational Type Soundness

First, let us consider what type soundness theorem should be satisfied by a gradual language. As discussed in Section 1, most previous work focuses on operational safety theorems: if a program runs to a value then it in some way satisfies its type [Greenman and Felleisen, 2018] and if it raises blame the blame is directed toward a dynamically typed component [Tobin-Hochstadt and Felleisen, 2008, Wadler and Findler, 2009].

However these soundness theorems are not *directly* useful for the purposes of program optimization. Since the language's operational semantics is only defined for a closed program, and a type safety theorem is only defined in terms of the operational semantics, there is a large formal gap to proving that it is safe to perform an optimization. Because of this, almost none of the previous work on gradually typed languages shows that any type-based optimizations or refactorings are valid, despite this being one of the purported purposes of gradual type casts.

Instead I propose that terms in a gradually typed language should satisfy the same *relational* soundness theorems that hold in an *effectful* typed language with the same sort of types. What these principles are depends on the type structure, but many can be formulated as *contextual equivalences* of programs. Two programs are considered contextually equivalent when they are indistinguishable in the context of any larger program. This means that one can be swapped out for another with no observable change in program behavior. I define contextual equivalence as follows:

Definition 1 (Contextual Equivalence)**.** *Two terms t and u of the same type are* contextually equivalent*, written $t \approx^{ctx} u$ when for any closing context C, $C[t] \equiv C[u]$.*

Here by a *context* I mean a program with a "hole" in it that can be filled in with the sort of expression of interest. I say it is a *closing* context when once filled in, it can be executed as a whole program (for instance it does not contain unbound variables). Finally, I appeal to a notion of equivalence $\equiv$ of behavior for whole programs that necessarily differs for different languages depending on what effects they have. For instance, in a language whose only effect is divergence, $p_1 \equiv p_2$ would be defined to hold when either both programs diverge or both terminate. If the language also allowed for printing to a fixed standard output, then equivalence would also require that they print the same message in addition to cotermination, etc.

I prefer soundness theorems stated in terms of contextual equivalence (and later contextual approximation), because they provide *directly usable* information to the programmer or optimizer: if two terms are known to be contextually equivalent, then one is a possible refactoring or optimization of the other. In particular, because intensional information such as running time or space usage is typically *not* considered observable, this can justify the replacement of a less efficient program with a more efficient one.

The soundness theorems themselves are specific to the type structure being studied. In this dissertation I will focus on the soundness theorems for simple types: products, sums and functions, but also on

the more complex parametricity theorems for polymorphic types. For simple types, the soundness theorems of interest are the soundness of appropriate β and η laws for contextual equivalence. In Figure1, I present call-by-value β, η laws for functions and booleans, two connectives I will use as examples throughout the document. The β laws should be uncontroversial: they are typical operational reductions stated as contextual equivalences, for instance restricting the input of a function to be a value before reducing. In fact, these β laws are contextual equivalences in many *dynamically typed* languages, and so do not say much about type soundness.

The η laws however are not satisfied by dynamic languages, and so can be thought of as *the difference* between reasoning about dynamically typed programs and statically typed programs. The general idea of an η law is that it says that values[1] of each type are all equivalent to a value produced by an introduction form of that type, and all behavior they exhibit is given by the elimination forms for that type.

For example, the η law for functions says that any value of function type is equivalent to one produced by λ by simply applying the original value to the λ-bound variable. If v is a λ function itself, this follows already by β equivalence, but the more useful case is when v is *itself* a variable. This shows that the η principle is really about restricting the power of the context to distinguish between programs: if every variable at function type is equivalent to a λ, that means that the context cannot pass non-function values as inputs.

The η law for strict pairs is of a dual nature: rather than being about terms whose output type is $A_1 \times A_2$, it is concerned with terms who have a free variable of type $A_1 \times A_2$. In plain English, it says that any term with a free variable $y : A_1 \times A_2$ is equivalent to one that immediately pattern-matches on y, and then replaces y with the reconstructed pair. This means first and foremost that pattern-matching on y is *safe*: compare to a strongly typed dynamic language where projecting from the pair would result in a runtime type error on non-pair inputs like booleans or functions. Second, it means that there is no more to a value of product type than what its components are, since the value in the continuation of the pattern match is replaced by the reconstructed pair. This rules out things like lazy evaluation, where a pair is a thunk that is forced when a pattern match is performed, and also rules out "junk" in the value like runtime reflection to determine the origin of the value.

These η principles provide a principled foundation for type-based optimization and refactoring. For instance, suppose you have an input $y : A_1 \times A_2$ and construct a function that pattern matches on x: $\lambda z : B.\texttt{let}\ (x_1, x_2) = y; t$, Upon reviewing the code you might decide that it is clearer to perform the pattern-match before constructing the function, perhaps because the second component of the pair is never used, and you want to reduce the size of the closure at runtime, rewriting it as $\texttt{let}\ (x_1, x_2) = y; (\lambda z : B.t)$. This intuitively obvious optimization is justified by the $\beta\eta$ principles for pairs:

$$\begin{aligned}
\lambda z : B.\texttt{let}\ (x_1, x_2) = y; t &\approx^{\text{ctx}} \texttt{let}\ (x_1', x_2') = y; \lambda z : B.\texttt{let}\ (x_1, x_2) = (x_1', x_2'); t && (\eta\times)\\
&\approx^{\text{ctx}} \texttt{let}\ (x_1', x_2') = y; \lambda z : B.t[x_1'/x_2][x_2'/x_1] && (\beta\times)\\
&\approx^{\text{ctx}} \texttt{let}\ (x_1, x_2) = y; \lambda z : B.t && (\alpha)
\end{aligned}$$

First, the η principle says you can deconstruct x immediately, and then the β principle simplifies the continuation, and α equivalence fixes the change of variable names. This program equivalence would not be valid in a dynamic language because it might introduce a runtime error into the program if y is instantiated with a non-pair, you are only justified in making the optimization because the type information $y : A_1 \times A_2$ is reliable.

This might look difficult to get wrong, but this transformation is invalid in the transient semantics of gradual typing [Vitousek et al., 2017]. In transient semantics, only the top-level constructor gives reliable information, with deeper checks performed only as a value is inspected. For instance, in transient semantics, the type $\texttt{bool} \times \texttt{number}$ includes values like $(\lambda x.x, \texttt{true})$, which is a pair, but the components of the pair do not match the type. Then η expansion for pairs is not generally valid, because pattern matching on a value like this produces a type error: $\texttt{let}\ (y, z) = (\lambda x.x, \texttt{true}); (y, z)$ runs to a type error reporting that either $\lambda x.x$ is not a boolean or that $\texttt{true}$ is not a number. This limits the flexibility of type-based optimization and refactoring.

For more advanced type features, $\beta\eta$ equations are not the only contextual equivalences that should be satisfied. For polymorphism, the type soundness theorem of interest is that of *parametricity* [Reynolds, 1983], which states that the behavior of polymorphic function is in a precise sense independent of the type

[1] in call-by-value languages η applies to values, but pure languages have η for all terms and for call-by-name languages η is for *strict* terms

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda x.t)\, v \approx^{\text{ctx}} t[v/x] : B} \to\beta \qquad \frac{\Gamma \vdash v : A \to B}{\Gamma \vdash v \approx^{\text{ctx}} \lambda x.v\, x : A \to B} \to\eta$$

$$\text{RIGHT}{=}\times\beta \quad \texttt{let}\ (x_1, x_2) = (v_1, v_2); t \approx^{\text{ctx}} t[v_1/x_1][v_2/x_2] \qquad \frac{y : A_1 \times A_2 \in \Gamma \qquad \Gamma \vdash t : B}{\Gamma \vdash t \approx^{\text{ctx}} \texttt{let}\ (x_1, x_2) = y; t[(x_1, x_2)/y] : A} \times\eta$$

Figure 1: Call-by-value β, η Laws for Functions and Strict Pairs

$$A \sqsubseteq ? \qquad A \sqsubseteq A \qquad \frac{A \sqsubseteq B \sqsubseteq C}{A \sqsubseteq C} \qquad \frac{A \sqsubseteq A' \qquad B \sqsubseteq B'}{A \to B \sqsubseteq A' \to B'}$$

Figure 2: Type Precision

that instantiates it. This theorem is also typically stated in terms of contextual equivalences, and provides programmers with a powerful reasoning tool about the expressiveness of programs at polymorphic types.

3.2 Graduality

While $\beta\eta$ equations and other contextual equivalence theorems ensure that the types are *useful*, the *graduality* theorem ensures there is a "smooth transition" from dynamic to static typing. By this I mean that a gradual language should support a certain pattern of program development where (partially) dynamically typed programs are gradually annotated/rewritten to be more strongly typed. The graduality theorem, also known as the gradual guarantee, says that changing types to be more precise should only result in very restricted changes in behavior: either the more precise program has the same behavior, or raises a type error.

To formalize this property, I first have to define what is meant by an increase in *precision* in types, which is defined by the standard type precision ordering $\sqsubseteq$, given in Figure 2, where $A \sqsubseteq B$ is read as "A is more precise than B". I then define a similar *term precision* ordering by saying that $t \sqsubseteq t'$ holds when t and t' have the same type erasure and every type annotation in t is more precise than the corresponding annotation in t'. The graduality theorem then says that if $t \sqsubseteq t'$ then the behavior of t refines t'. I formalize this using the following notion of *contextual error approximation*, an ordering similar in spirit to contextual equivalence.

Definition 2 (Contextual Error Approximation, Term Precision)**.** *Error approximation is defined in two stages.*

- *First, if t, t' have the same type and context, then define $t \sqsubseteq^{ctx} t'$ to hold when for any closing context C, $C[t] \preceq C[t']$.*
- *Second, if $\Gamma \vdash t : A, \Gamma' \vdash t' : A'$ and $\Gamma \sqsubseteq \Gamma'$ and $A \sqsubseteq A'$, define $t \sqsubseteq\!\!\approx t'$ to hold when $t \sqsubseteq^{ctx} \langle A \swarrow A' \rangle t'[\langle \Gamma' \nwarrow \Gamma \rangle]$ where*

$$\langle \cdot \nwarrow \cdot \rangle = \emptyset \qquad \langle \Gamma_2, x : A_2 \nwarrow \Gamma_1, x : A_1 \rangle = \langle \Gamma_2 \nwarrow \Gamma_1 \rangle, x \mapsto \langle A_2 \nwarrow A_1 \rangle x$$

Here I have parameterized the definition of contextual error approximation by a notion of error approximation for closed programs which I write $\preceq$. Similarly to contextual equivalence, this will be dependent on what effects are present in the language. For a language with only divergence and errors, I say $p \preceq p'$ holds when either p runs to an error, or both terms terminate or both terms diverge. If the language allowed printing, I would need to say that everything printed before p errors must be a prefix of what is printed by p', and similarly modify for other effects. Note that this ordering provides a formal foundation for Findler-Blume-Felleisen's notion of error projection mentioned in Section2.2.

$$
\begin{array}{rcl}
\text{types } A, B & ::= & ? \mid \texttt{bool} \mid A \to B \mid A \times B \\
\text{ground types } G & ::= & \texttt{bool} \mid ? \to ? \mid ? \times ? \\
\text{terms } t, u & ::= & \langle A \Leftarrow B \rangle t \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\,\{t\}\{t\} \\
& & \mid \lambda x : A.t \mid t\,u \mid (t, u) \mid \texttt{let}\ (x, y) = t; u
\end{array}
$$

Figure 3: Syntax of Cast Calculus

4 From Embedding-Projection Pairs to Relational Soundness Theorems

Next, I show how an interpretation of types as embedding-projection pairs gives a simple model of gradual typing that underpins existing cast semantics and enables the proof of the relational soundness theorems proposed in Section 3. I work out the example of a simple call-by-value gradual cast calculus supporting booleans and functions. This section is based on New and Ahmed [2018], which includes sum types as well.

4.1 The Cast Calculus

Our starting point is a standard gradual cast calculus based on Wadler and Findler [2009], that serves as a target for elaboration from a gradual surface language. Figure 3 contains the syntax of types and terms in the calculus. The language is essentially a typed language extended with the dynamic type ? and the cast form $\langle A \Leftarrow B \rangle M$ which casts the output of a term $M : B$ to the type A, possibly erroring if it does not meet the type A. Since this is a cast calculus and not a surface language, the type system is a standard type system, extended with the cast rule:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \langle A \Leftarrow B \rangle t : B}$$

More important is the *operational semantics* of casts. Again, most of it is standard call-by-value operational semantics, so I describe only the reductions for *casts* in Figure 4. The first rule, DynDyn says the cast from ? to itself is the identity. The next rule, TagUp breaks down large casts to ? into a cast to a "tag" type and then a structural cast. What the tag types are depends on the language, but usually there is one for each typed connective. For our calculus, the tag types consist of our base type `bool`, and also our connectives applied to the dynamic type: $? \times ?$ and $? \to ?$. Next, define $\lfloor A \rfloor$ to be a function that extracts the tag type of a type. So $\lfloor \texttt{bool} \rfloor = \texttt{bool}$, $\lfloor A_1 \times A_2 \rfloor = ? \times ?$ and $\lfloor A \to B \rfloor = ? \to ?$, and $\lfloor ? \rfloor$ is undefined.

The next rule, TagDn is the dual of TagUp, applying to complex casts *out of* the dynamic type. The next two rules, TagMatch and TagMismatch formalize the simple tag checking that takes place in dynamically typed languages. When a tagged value $\langle ? \Leftarrow G' \rangle v$ is projected to a type G, the tags are checked for equality. If the tags match (TagMatch), the value is projected out, but if the tags differ (TagMismatch) an error is raised. Next the product rule ProdCast says that a cast between product types proceeds by casting each component of the pair. The final rule FunCast is the rule for casts between two function types, familiar from Findler and Felleisen [2002]. It produces a new function that when applied to a value, casts it to the original input type, and then casts back the output.

4.2 Semantics by Elaboration

How can one prove that the cast calculus satisfies relational type soundness and graduality? In the remainder of this section I give a method that proves these theorems by uncovering the semantic infrastructure that powers the semantics of casts.

Our first goal is to satisfy the relational type soundness criteria. For simple types, this means the validity of the CBV $\beta\eta$ equations listed in Figure 1. To ensure that these are satisfied, I give semantics of our cast calculus by elaboration to a statically typed language where these $\beta\eta$ laws are satisfied. This means that the casts of our calculus must be implementable in our typed language, so in particular our typed language must support some notion of runtime error, but only when the programmer explicitly invokes it. So I use a simply typed call-by-value calculus, which in order to interpret the gradual language

$$E[\langle ? \Leftarrow ? \rangle v] \mapsto E[v] \textsc{ DynDyn}$$

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

$$\frac{G = G'}{E[\langle G \Leftarrow ? \rangle \langle ? \Leftarrow G' \rangle v] \mapsto E[v]} \textsc{ TagMatch} \qquad \frac{G \neq G'}{E[\langle G \Leftarrow ? \rangle \langle ? \Leftarrow G' \rangle v] \mapsto \mho} \textsc{ TagMismatch}$$

$$E[\langle A_1' \times A_2' \Leftarrow A_1 \times A_2 \rangle (v_1, v_2)] \mapsto E[(\langle A_1' \Leftarrow A_1 \rangle v_1, \langle A_2' \Leftarrow A_2 \rangle v_2)] \textsc{ ProdCast}$$

$$E[\langle A' \to B' \Leftarrow A \to B \rangle v] \mapsto E[\lambda x : A'. \langle B' \Leftarrow B \rangle (v(\langle A \Leftarrow A' \rangle x))] \textsc{ FunCast}$$

Figure 4: Cast Calculus Operational Semantics

$$\begin{array}{llll} \text{types} & A, B & ::= & \mu X.A \mid X \mid A + B \mid \texttt{bool} \mid A \to B \mid A \times B \\ \text{terms} & t, u & ::= & \mho \mid \text{roll}_{\mu X.A} t \mid \text{unroll}_{\mu X.A} t \mid \texttt{case}\, t\, \{\texttt{inl}\ x_l.u_l \mid \texttt{inr}\ x_r.u_r\} \\ & & & \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\, t\, \{t\}\{t\} \mid \lambda x : A.t \mid t\, u \mid (t, u) \mid \texttt{let}\ (x, y) = t; u \end{array}$$

Figure 5: Simply Typed Metalanguage

will also need (1) recursive types, (2) sum types and (3) an explicit runtime error I write as $\mho$. The syntax and operational semantics of this language are given in Figure 5.

Next, I need to model the types of the cast calculus as static types. I define a function $|\cdot|$ that maps gradual types to static types. For simple types, the same syntactic types suffice $|\texttt{bool}| = \texttt{bool}$, $|A_1 \times A_2| = |A_1| \times |A_2|$ and $|A \to B| = |A| \to |B|$, ensuring that $\beta\eta$ equations hold by inheriting them from the metalanguage. Then, following classical models of dynamic typing in domain theory [Scott, 1976], interpret the dynamic type as a recursive sum type of the "tags": booleans and functions.

$$|?| = \mu X.\texttt{bool} + (X \times X) + (X \to X)$$

With these definitions, it is easy to define an elaboration of the cast calculus into our typed language for all cases *except* the casts.

How should the behavior of casts be defined? Rather than attempt to directly implement them following the operational semantics, it would be preferable to specify the casts for each type independently, with the caveat that interpretation of the dynamic type was designed with casts in mind. This makes the semantics more modular: you don't need to consider every possible combination of types. To do this, certain specific casts will play a key role: the casts to the dynamic type $\langle ? \Leftarrow A \rangle$ and from the dynamic type $\langle A \Leftarrow ? \rangle$. The reasoning is that in a gradual language, all cast behavior should reduce to the behavior of these casts by the equivalence

$$\langle B \Leftarrow A \rangle t \approx^{\text{ctx}} \langle B \Leftarrow ? \rangle \langle ? \Leftarrow A \rangle t.$$

Intuitively, this should be true because the cast $\langle B \Leftarrow A \rangle t$ should enforce Bs type on an A term, but in a coherent semantics this should have the same meaning as first forgetting that something satisfies A by casting to dynamic, and then enforcing Bs type.

With this intuition, define a *semantic gradual type*, i.e., a possible denotation of a gradual type relative to our choice of metalanguage and interpretation of the dynamic type $|?|$, as a choice of implementation type and an interpretation of the casts $\langle ? \Leftarrow A \rangle$ and $\langle A \Leftarrow ? \rangle$. The property needed is that these casts form an *embedding-projection pair*

Definition 3. *An* embedding-projection pair *from A to B, written $(e, p) : A \lhd B$ is a pair of functions $e : A \to B$ (the embedding) and $p : B \to A$ (the projection) satisfying two properties:*

- *(*Retraction*)* $x : A \vdash x \approx^{ctx} p(e(x)) : A$
- *(*Projection*)* $y : B \vdash e(p(y)) \sqsubseteq^{ctx} y : B$

$$
\begin{aligned}
e_? &= \lambda x.x \\
p_? &= \lambda x.x \\
e_{\texttt{bool}} &= \lambda x : \texttt{bool}.\texttt{inl}\ x \\
p_{\texttt{bool}} &= \lambda d : |?|.\texttt{case}\ d\,\{\texttt{inl}\ x.x \mid \texttt{else}\ \mho\} \\
e_{A_1 \times A_2} &= \lambda p : |A_1| \times |A_2|.\texttt{inl inr}\ (e_{A_1}(\pi_1 p), e_{A_2}(\pi_2 p)) \\
p_{A_1 \times A_2} &= \lambda d : |?|.\texttt{case}\ d\,\{\texttt{inl inr}\ p.(p_{A_1}(\pi_1 p), p_{A_2}(\pi_2 p)) \mid \texttt{else}\ \mho\} \\
e_{A \to B} &= \lambda f : |A| \to |B|.\texttt{inr inr}\ (\lambda d : |?|.e_B(f(p_A d))) \\
p_{A \to B} &= \lambda d : |?|.\texttt{case}\ d\,\{\texttt{inr inr}\ f.\lambda y : |A|.p_B(f(e_A y)) \mid \texttt{else}\ \mho\}
\end{aligned}
$$

Figure 6: Embedding-projection Pair Semantics of Gradual Types

The intuition behind the RETRACTION property is that embedding A into B should lose no information, so when projecting back down to A after embedding, the result should be equivalent to the original value. The intuition behind the PROJECTION property is similar to graduality: projecting down to A should result in a value that refines the behavior of the original program: it may error because B might have values that have no analogue in A, but otherwise should behave similarly. By embedding the output back into B, this can be formalized using contextual error approximation to provide a uniform notion of approximating behavior.

Then a semantic gradual type gives an interpretation of the type both as a static type and an interpretation of its casts by specifying an embedding-projection pair *to the dynamic type*, i.e., how its values are embedded in the dynamic type, and how to "check" that dynamically typed values are valid members of the type.

Definition 4. *A* semantic *gradual type A consists of a type $|A|$ and an embedding-projection pair $(e_A, p_A) : A \triangleleft |?|$.*

The gradual types and their semantics are given in Figure 6. The embedding and projection for the dynamic type are just the identity. For booleans, the embedding is just the injection and the projection pattern matches on the input: if it is a left injection, it returns the boolean, and otherwise it errors. The function embedding starts out the same way as the boolean embedding but using the right injection. However, we need to inject into $|?| \to |?|$ but we have a function $f : |A| \to |B|$. The solution is to compose with the *projection* for A and *embedding* $|B|$ to make a dynamic function. This wrapping of functions is familiar from the semantics of contracts, and from category theory: it is the action of $\to$ as a functor.

Proving that these form embedding-projection pairs is straightforward given some general technique for proving contextual equivalence and error approximation results. In the past, I have with my co-authros used standard techniques of step-indexed logical relations [New and Ahmed, 2018] in addition to providing sound inequational theories [New and Licata, 2018, New et al., 2019b]. Finally, there is a compositional translation $[\![\cdot]\!]$ from the gradual language to the typed metalanguage, which maps everything but the casts to the analogous form in the metalanguage, and translates $\langle A \Leftarrow B \rangle$ by routing the cast through the dynamic type:

$$[\![\langle A \Leftarrow B \rangle M]\!] = p_A(e_B([\![M]\!]))$$

Which implies in the cast calculus the program equivalence

$$\langle A \Leftarrow B \rangle M \approx^{\text{ctx}} \langle A \Leftarrow ? \rangle \langle ? \Leftarrow B \rangle M$$

4.3 From EP Pairs to Graduality

Next, we seek to prove the graduality theorem. Intuitively the behavior of the casts is the key to ensuring that graduality holds: the elaboration of a term with more precise types than another only differs in that it contains more precise casts. To prove this key property I expand our semantic analysis from gradual types to gradual type *precision*. The intuition is that when $A \sqsubseteq B$, A should have a similar relationship to B that it does to the dynamic type, since in particular $A \sqsubseteq ?$ for all A. I formalize this by requiring that the casts between A and B should *also* form an embedding-projection pair.

$$\frac{A \in \{\texttt{bool}, ?\}}{A \sqsubseteq A}\ \textsc{Refl} \qquad \frac{A \sqsubseteq G \qquad G \in \{\texttt{bool}, ? \to ?\}}{A \sqsubseteq ?}\ \textsc{Tag} \qquad \frac{A \sqsubseteq A' \qquad B \sqsubseteq B'}{A \to B \sqsubseteq A' \to B'}\ \textsc{Func}$$

Figure 7: Syntactic Type Precision

Definition 5 (Semantic Type Precision)**.** *Let A and B be semantic gradual types. Then A is* semantically more precise than *B, written $A \lesssim B$ when there exists an embedding projection pair $(e_{AB}, p_{AB}) : |A| \triangleleft |B|$:*

1. *(Embedding factorization) For every $v : |A|$, $e_B(e_{AB}v) \approx^{ctx} e_A v$.*
2. *(Projection factorization) For every $v : |B|$, $p_{AB}(p_B v) \approx^{ctx} p_A v$.*

These last two conditions ensure that the ep pair from A to B "agrees with" their respective ep pairs to ?, which formalizes the idea that these casts are enforcing the same property. Because embeddings are injective/projections are surjective, this embedding is unique up to $\approx^{\text{ctx}}$ if it exists, as is the projection.

Now, define the syntactic term precision judgment $A \sqsubseteq B$ with the goal of making a sound system for proving semantic term precision $A \lesssim B$. I give a parsimonious presentation of the precision rules in Figure 7. This presentation is easily seen to be equivalent in provability to the one given in Figure 2, but is structured better for the soundness proof, in that one can read these proofs as a *syntax* for constructing ep pairs.

Theorem 6 (Soundness of Type Precision)**.** *If $A \sqsubseteq B$ then $A \lesssim B$.*

The proof is by induction on syntactic precision proofs. The Refl case corresponds to a pair of identity functions, the Tag case corresponds to composition with the left and right injection ep pairs and the Func case to the higher order part of the function ep pair.

Finally, I define a semantic notion of *term precision* that formalizes the semantic property in the graduality theorem.

Definition 7 (Semantic Term Precision)**.** *Let $A_1, \ldots, B, A_1', \ldots, B'$ be semantic gradual types such that $A_1 \lesssim A_1', \ldots, B \lesssim B'$, and let $x_1 : A_1, \ldots \vdash M : B$ and $x_1' : A_1', \ldots \vdash M' : B'$. Then, M is semantically more precise than M', written $M \lesssim M'$, if*

$$\texttt{let}\ x_1' = e_{A_1 A_1'} x_1; \cdots M \sqsubseteq^{ctx} p_{BB'} M'$$

Then the graduality theorem can be reformulated as follows:

Theorem 8. *If $M \sqsubseteq M'$, then $[\![M]\!] \lesssim [\![M']\!]$.*

This is established by induction over the proof that $M \sqsubseteq M'$, using a logical relation that is sound with respect to $\sqsubseteq^{\text{ctx}}$. The cast cases in particular follow from the embedding-projection pair and factorization properties.

4.4 How to design a Gradual Language

While I have framed this section *analytically* as studying a specific gradual cast calculus and identifying structures that it contains, one can reverse the process to design new gradual languages by *synthetically* constructing them from the ep pair structures I have identified here. That is, one can start by choosing some rich metalanguage with types that satisfies our desired relational soundness criteria, and then designing a suitable universal type. Then one can define our gradual types by constructing ep pairs into this universal type, and then design a cast calculus with the theorems I have proven above as design goals. One can then define a gradual surface language using standard methods of defining consistency and consistent subtyping, utilizing techniques such as the AGT framework to design the type system, but elaborating to our cast calculus to define the operational semantics. I apply this methodology in Section 6.

5 Gradual Type Theory: Deriving Cast Semantics from Relational Soundness Theorems

In the previous section I have shown that the view of gradual types as embedding-projection pairs into the dynamic type helps to prove type soundness and graduality of a gradual language, i.e., that the theory of embedding-projection pairs is sufficient to develop the metatheory of some sound gradual languages. The goal of the work presented in this section is to study the converse: to what extent is the theory of embedding-projection pairs *inherent to* the study of sound gradually typed languages? Furthermore, how much is the design of gradually typed languages constrained by the requirements of $\beta\eta$ equality and graduality?

I argue that the embedding-projection pair semantics is intuitively justified by the requirements of the graduality theorem by giving an *axiomatic* formulation of the semantic term precision relation that underlies graduality. I add intuitive axioms like reflexivity, transitivity, compositionality and $\beta\eta$ equivalence for this relation. Finally, I add soundness and completeness rules for casts, which are intuitively justified by the graduality property, and have appeared in simulation proofs of graduality [Siek et al., 2015]. I call this axiomatic formulation *gradual type theory*.

One can then derive using gradual type theory that when $A \sqsubseteq B$, the casts between A and B form a Galois connection, a mild weakening of the embedding-projection pair property. Furthermore, one can *derive* most of the cast semantics presented in Section 4 from $\beta\eta$ equality and the soundness and completeness principles for casts. Since soundness and completeness define the casts up to contextual equivalence, this shows that the derived rules are the only possible rules to satisfy $\beta\eta$ and graduality, showing that these form very strong design constraints indeed.

Furthermore, to show the generality of the theory, I provide a type theory capable of encoding multiple evaluation orders, enabling the derivation of cast semantics for call-by-value, call-by-name and mixes of the two, like Haskell's lazy semantics. I do this by building on Levy's call-by-push-value calculus [Levy, 2003]. The cast semantics I derive for the lazy semantics reproduces previous results from the literature on lazy contracts [Degen et al., 2012].

This section is primarily based on New et al. [2019b], which builds on similar work for a CBN calculus published in New and Licata [2018]. These works are based on a categorical semantics of gradual typing described in New and Licata [2018], which is heavily based on the mathematical work in Shulman [2008].

5.1 Syntax of Gradual Type Theory

In order to accommodate both call-by-name (CBN) and call-by-value (CBV) languages, I base the syntax of gradual type theory (GTT) on Levy's *call-by-push-value* (CBPV) language, a simple core calculus into which both call-by-value and call-by-name languages can be embedded in such a way that preserves the $\beta\eta$ equivalences of each [Levy, 2003]. These embeddings are analogous to CPS translations, and enable the study of both CBV and CBN simulateneously by considering the fragments of CBPV that contain them. A key feature of CBPV that supports this is that it distinguishes between effectful computations, written M, from pure values V (familiar from CBV)[2]

Like CBPV, GTT has two kinds of types: value types written A, which like the types in a CBV language classify the shape of data, and computation types, written B, which like CBN types classify *behavior* of terms. Values are typed as $\Gamma \vdash V : A$ where contexts Γ only contain variables with value types. Computations $\Gamma \vdash M : B$ also have a context Γ of value types, but their output type is a computation type. The intuition is that an effectful term M when run, is supplied with values as given by Γ and then behaves as given by B.

The relevant fragment of GTT types is given in Figure 8. The value types include the dynamic type ?, booleans `bool`, and pairs of values $A_1 \times A_2$ and also *thunks* Thunk B that behave as a B when forced. The computation types used are the function type $A \to B$ which classifies computations that receive A values and behave as B computations, and Returns A which classifies computations that perform effects, and if they return, return A values. Since in CBV, all terms return values, the embedding of CBV into GTT translates the CBV function type $A \to_{cbv} A'$ as Thunk $(A \to \text{Returns } A')$: a CBV function is a thunk that when it receives an A value, performs effects and possibly returns A' results.

Next, extend the CBPV type and term structure to include *precision* relations: value type precision, computation type precision and corresponding value and computation precision judgments. These are similar to the previous precision judgments, noting that one can only compare like for like: value types

[2]CBPV also distinguishes "strict"/"linear" S called stacks that are needed to formulate CBN η equality, but I elide this here for space reasons.

$$
\begin{array}{rrl}
\text{value types} & A & ::= ? \mid \texttt{bool} \mid \ \mid A_1 \times A_2 \mid \text{Thunk } B \\
\text{computation types} & B & ::= A \to B \mid \text{Returns } A \\
\text{values} & V & ::= \langle A' \leftharpoonup A \rangle V \mid \texttt{true} \mid \texttt{false} \mid \ \mid (V, V) \mid \text{thunk}\{M\} \\
\text{computations} & M, N & ::= \langle B \leftharpoondown B' \rangle M \mid \texttt{if } V \{M\}\{N\} \mid \text{force } M \mid \texttt{let } (x, y) = V; M \\
& & \mid \lambda x : A.M \mid MV \mid \text{ret } V \mid x \leftarrow M; N
\end{array}
$$

Figure 8: GTT Types and Terms

$$
\frac{G \in \{\texttt{bool}, ?, \text{Thunk } (? \to \text{Returns } ?)\}}{G \sqsubseteq ?} \qquad \texttt{bool} \sqsubseteq \texttt{bool} \qquad \frac{A \sqsubseteq A' \sqsubseteq A''}{A \sqsubseteq A''} \qquad \frac{B \sqsubseteq B' \sqsubseteq B''}{B \sqsubseteq B''}
$$

$$
\frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 \times A_2 \sqsubseteq A_1' \times A_2'} \qquad \frac{B \sqsubseteq B'}{\text{Thunk } B \sqsubseteq \text{Thunk } B'} \qquad \frac{A \sqsubseteq A'}{\text{Returns } A \sqsubseteq \text{Returns } A'} \qquad \frac{A \sqsubseteq A' \qquad B \sqsubseteq B'}{A \to B \sqsubseteq A' \to B'}
$$

Figure 9: GTT Type Precision

can be more or less precise than other value types, but are not directly comparable to computation types. The value and computation type precision are presented in Figure 2, and are similar to the previous type precision. The dynamic type is a value type, and is the least precise such value type. I restrict the ?-top rule to only include the ground types used in Section 4, but using transitivity, $A \sqsubseteq ?$ is derivable for all CBPV translation of all types from that section.

Next, I show a fragment of the rules for term precision in Figure 10. The first rules give reflexivity and transitivity. The next two give two substitution principles, one for substituting values into values and one for substituting values into terms.

Next, I add $\beta\eta$ rules to the logic, where $\sqsupseteq\sqsubseteq$ means there is a rule for each of $\sqsubseteq$ and $\sqsupseteq$, for space saving reasons, only the cases for functions are shown. When combined with the elaboration of CBV/CBN into CBPV, these $\beta\eta$ rules faithfully reflect on the restricted $\beta\eta$ principles of CBV/CBN surface languages.

The last four rules in the figure are the most important as they specify the behavior of casts. Each of upcasts and downcasts gets a pair of rules, one called soundness and the other, completeness. The soundness rule for upcasts says that casting a value up from A to A' results in a larger program in the term ordering. The completeness theorem says that this is the *minimal* such value, i.e., it has the least possible amount of behavior added in order to satisfy the type A'. This gives a specification for how the upcast should be implemented: it needs to construct the *minimal* program whose behavior is refined by the input. The downcast gets a dual specification. The soundness rule says the result of a downcast should refine the behavior of the original. Since making something smaller includes errors, this

$$
V \sqsubseteq V \qquad \frac{V \sqsubseteq V' \sqsubseteq V''}{V \sqsubseteq V''} \qquad M \sqsubseteq M \qquad \frac{M \sqsubseteq M' \sqsubseteq M''}{M \sqsubseteq M''}
$$

$$
\frac{V_i \sqsubseteq V_i' \qquad V_o \sqsubseteq V_o'}{V_o[V_i/x] \sqsubseteq V_o'[V_i'/x]} \qquad \frac{V \sqsubseteq V' \qquad M \sqsubseteq M'}{M[V/x] \sqsubseteq M'[V'/x]} \qquad (\lambda x : A.M)V \sqsupseteq\sqsubseteq M[V/x] \qquad \frac{M : A \to B}{M \sqsupseteq\sqsubseteq \lambda x : A.Mx}
$$

$$
\cdots
$$

$$
\frac{V : A \qquad A \sqsubseteq A'}{V \sqsubseteq \langle A' \leftharpoonup A \rangle V} \ \textsc{UpSound} \qquad \frac{V \sqsubseteq V' : A \sqsubseteq A'}{\langle A' \leftharpoonup A \rangle V \sqsubseteq V'} \ \textsc{UpComplete}
$$

$$
\frac{M : B \qquad B' \sqsubseteq B}{\langle B' \leftharpoondown B \rangle M \sqsubseteq M} \ \textsc{DnSound} \qquad \frac{M' \sqsubseteq M : B' \sqsubseteq B}{M' \sqsubseteq \langle B' \leftharpoondown B \rangle M} \ \textsc{DnComplete}
$$

Figure 10: GTT Term Precision (fragment)

includes the possibility the cast errors if there are no non-error terms that refine the term's behavior. The completeness says that the downcast should be the *maximal* such term, i.e., the one that errors the *least*.

The soundness theorems here are very similar in spirit to the original statement of the gradual guarantee: that adding certain casts should result in a refinement of program behavior. The completeness theorems extend this to be a kind of *optimality* theorem: the downcasts add errors, but no more than necessary, and the upcasts add behavior, but as little as possible.

5.2 Consequences of Gradual Type Theory

I now summarize the main theorems that are *provable* in gradual type theory. Most of these theorems are in the form of *order-equivalences* $M \sqsupseteq\sqsubseteq M'$ between computations, and which represent *contextual equivalences* between M and M' that are necessary consequences of the axioms of gradual type theory, i.e., consequences of relational type soundness and graduality.

First, one can show that almost all of the lemmas stated in Section 4 as being key lemmas to *proving* graduality, are in fact *necessary*.

Theorem 9 (Some Consequences of Gradual Type Theory)**.**

- *If* $A \sqsubseteq A' \sqsubseteq A''$*, then* $\langle A'' \nwarrow A' \rangle \langle A' \nwarrow A \rangle V \sqsupseteq\sqsubseteq \langle A'' \nwarrow A \rangle V$
- *If* $B \sqsubseteq B' \sqsubseteq B''$*, then* $\langle B \swarrow B' \rangle \langle B' \swarrow B'' \rangle M \sqsupseteq\sqsubseteq \langle B \swarrow B'' \rangle M$
- *(Embedding) If* $A \sqsubseteq A'$*, then* $x \leftarrow \langle FA \swarrow FA' \rangle M; \mathit{ret}\, \langle A' \nwarrow A \rangle x \sqsubseteq M$
- *(Projection) If* $A \sqsubseteq A'$*, then* $\mathit{ret}\, V \sqsubseteq \langle FA \swarrow FA' \rangle \mathit{ret}\, \langle A' \nwarrow A \rangle V$

Items 1 and 2 of the theorem are the factorization properties from the definition of semantic type precision. Item 3 is the CBPV rendering of the embedding property of an ep pair, while item 4 is the rendering of one direction of the projection property.

Furthermore, one can show that for each connective, assuming the η principle of the type enables the derivation of the semantics of its casts from first principles. For instance, for functions, one can prove

$$\langle A \to B \swarrow A' \to B' \rangle M \sqsupseteq\sqsubseteq \lambda x : A. \langle B \swarrow B' \rangle (M \langle A' \nwarrow A \rangle x)$$

From the soundness/completeness theorems and η equivalence for functions.

6 Combining Graduality with Data Abstraction

To show the utility of the theory for embedding-projection pairs for designing new gradual languages, I design a new gradual language that supports a form of parametric polymorphism. This combination has proven notoriously difficult: attempts to develop gradually typed languages with parametric polymorphism have all either failed to preserve relationally parametric reasoning or failed to satisfy the graduality theorem [Ahmed et al., 2011, 2017]. Recent work has gone so far as to conjecture that parametricity and the graduality theorem are inherently incompatible [Toro et al., 2019]. To avoid the problems of previous work, I start with a semantic analysis and work backwards, developing a new syntax based on the type generation schemes previous work has used, rather than trying to apply novel semantics to a pre-existing static language. The work presented in this section has been submitted to a conference, but is available on my personal website [New et al., 2019a].

6.1 Why Gradual Parametricity Fails

Let us consider the "obvious" extension of a gradual language to include explicit polymorphism in the style of System F. One would add a new type $\forall X.A$, and introduction and elimination forms

$$\frac{\Gamma, X \vdash t : A}{\Gamma \vdash \Lambda X.t : \forall X.A} \qquad \frac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t[B] : A[B/X]} \qquad \frac{\Gamma \vdash t : ?}{\Gamma \vdash t[B] : ?}$$

Equipping this with the usual substitution-based semantics of polymorphism, however does not preserve parametricity due to the presence of dynamic casts. For instance consider the following function t_{np} for non-parametric:

$$t_{np} = \Lambda X. \lambda x : X. (x :?) : \mathtt{bool}$$

$$\frac{\Gamma, x : \text{Case } A \vdash t : B}{\Gamma \vdash \texttt{newcase}_A\ x; t : B} \qquad \frac{\Gamma \vdash t : \text{Case } A \qquad \Gamma \vdash u : A}{\Gamma \vdash \texttt{inj}_t\ u : \text{OSum}}$$

$$\frac{\Gamma \vdash t : \text{OSum} \qquad \Gamma \vdash u : \text{Case } A \qquad \Gamma, x : A \vdash t_c : B \qquad \Gamma \vdash t_e : B}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ u\{\texttt{inj}\ x.t_c \mid t_e\} : B}$$

Figure 11: Extensible Sum Type and First Class Cases

t_{np} is a polymorphic function of type $\forall X.X \to \texttt{bool}$ that takes an input of an abstract type X and then casts that input to `bool`. It casts through the dynamic type because the type system won't directly allow a cast from X to `bool`, but will allow any cast where one side is ?. In System F, by parametricity, any value of type $\forall X.X \to \texttt{bool}$ must be a constant function, because no information can be extracted from a value of abstract type X. However, in the naïve gradual extension, if you apply this function to `bool`, then the cast succeeds, and returns whatever the input was:

$$t_{np}[\texttt{bool}]b \mapsto^* (\lambda x : \texttt{bool}.(x : ?) : \texttt{bool})\texttt{true} \mapsto^* b$$

which violates our intuitive parametric reasoning.

I analyze the problem based on the semantic model I presented in Section 4. There, I gave a semantics to each gradual type A as having a static type interpretation $|A|$ and an embedding-projection pair to ?. So what should be the definition of $|\forall X.A|$? In the source language, the X ranges over *gradual types*, so based on our semantic analysis of gradual types, this would mean that intuitively one needs to quantify over *both* a static type and an embedding projection pair from that type into ?, which suggests the definition:

$$|\forall X.A| = \forall X.(X \lhd ?) \to |A|$$

where $X \lhd ?$ is some type encoding embedding-projection pairs, for example it can be encoded in the type $(X \to ?) \times (? \to X)$. This type can be used to encode the naïve semantics above. First, extend the dynamic type to include a case for polymorphic functions:

$$|?| = \mu D.\texttt{bool} + (D \times D) + (D \to D) + (\forall X.(X \lhd D) \to D)$$

Then the interpretation of type application is to pass in the ep pair associated to the type being instantiated:

$$[\![t[B]]\!] = [\![t]\!][|B|](e_B, p_B)$$

The benefit of our semantic approach is it now becomes very clear why the semantics is not parametric: the parametricity theorem one gets is only as strong as the type to which it translates, and by passing in an ep pair, the theorem one gets is very weak. Considering the earlier example, the interpretation of the type of t_{np} is

$$|\forall X.X \to \texttt{bool}| = \forall X.(X \to ?) \times (? \to X) \to X \to \texttt{bool}$$

and this type has quite a weak parametricity theorem associated to it, since one can simply embed the X into ? and then pattern match on ? to see if it is the boolean case and extract it.

6.2 Fresh Type Generation, Semantically

The problem is that gradual types must support an embedding-projection pair into the dynamic type, but the implementation of the dynamic type I have used so far exposes all information about its values. In order for any kind of parametricity theorem to hold, one must be able to have values of the dynamic type that are somehow inaccessible to certain parts of the program. This can be accomplished by generalizing the dynamic type from a static sum of a few ground cases, to a *dynamically extensible* sum type where new cases can be allocated at anytime that are "secret" to other parts of the program. This feature is similar to the extensibility of the ML exception type, and Racket's opaque struct mechanism provides similar functionality. On the theoretical side, it is equivalent to a typed formulation of the untyped sealing primitives in Sumii and Pierce [2004].

I give typing rules for this feature in Figure 11. I introduce two new types. OSum is the "open sum type", which can be dynamically extended with new cases. Case A is the type of first-class cases of the

$$\frac{X \cong A \in \Gamma \qquad \Gamma \vdash t : A' \qquad A' \lesssim A}{\Gamma \vdash \mathtt{seal}_X t : X} \qquad \frac{X \cong A \in \Gamma \qquad \Gamma \vdash t : A' \qquad A' \lesssim X}{\Gamma \vdash \mathtt{unseal}_X t : A} \qquad \frac{\Gamma, X \vdash t : A}{\Gamma \vdash \Lambda X.t : \forall^\nu X.A}$$

$$\frac{\Gamma \vdash t : \forall^\nu X.A \qquad \Gamma \vdash B \qquad \Gamma \vdash C \qquad \Gamma, Y \cong B, x : A[Y/X] \vdash u : C}{\Gamma \vdash \mathtt{let}\ x = t\,\{Y \cong B\}\mathtt{in}\ u : C}$$

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash B \qquad \Gamma \vdash C \qquad \Gamma, Y \cong B, x : ? \vdash u : C}{\Gamma \vdash \mathtt{let}\ x = t\,\{Y \cong B\}\mathtt{in}\ u : C}$$

Figure 12: Sealing and Fresh Universal Quantifiers

open sum type which inject A values. There are three forms for manipulating these values. First, the new-case form allows for the creation of fresh cases of the open sum type. Second, given a Case A and an A, one can inject it into the open sum type OSum. Finally, given a Case A, one can pattern match on an OSum value to see if it uses that case. The pattern match includes a continuation for if the case matches, and another catch-all case if it does not. Combined, the injection and pattern matching forms encode the semantic intuition that for each v : Case A, there is an isomorphism between OSum and $A + (\text{OSum} - v)$ for some type $\text{OSum} - v$ that cannot be represented without adding complexity to the language[3].

6.3 A Syntax for Fresh Gradual Polymorphism

Bringing it back to parametricity, I update our semantics of the dynamic type and polymorphic types,

$$|?| = \mu D.\mathtt{bool} + (D \times D) + (D \to D) + (\forall X.\text{Case } X \to D) + \text{OSum}$$
$$|\forall X.A| = \forall X.\text{Case } X \to |A|$$

The dynamic type now contains, in addition to the statically allocated cases from before, all of the dynamically allocated cases in OSum[4]. Next, the polymorphic type, instead of taking an arbitrary ep pair, is given a Case X, with which one can construct an ep pair from X to OSum, and therefore $|?|$. This rules out the behavior above where you can cast from X to `bool` by inspecting the dynamically typed output of the embedding, because the Case X cannot possibly be the same as the case for `bool`.

With some work, previous attempts for gradual polymorphic semantics can be encoded using these type interpretations or slight variations. However, these previous attempts are fairly complex to explain and have the unfortunate side effect of violating one or both of graduality and parametricity. Instead, I construct a new *surface syntax* for a "fresh" universal quantifier, based on the semantic ideas I have outlined.

I give the typing rules for fresh universal quantification in Figure 12. First, note that the context Γ includes not just term variables $x : A$, but also *abstract* type variables X, which behave like type variables in System F and *known* type variables $X \cong A$, which behave more like nominal type definitions. Under the assumption that $X \cong A$, one knows that X and A are *isomorphic* types, and so I give syntax to this isomorphism as $\mathtt{seal}_X$ and $\mathtt{unseal}_X$. Next, I add a fresh polymorphic type $\forall^\nu X.A$, using an annotation to note that this is not the ordinary universal quantifier. The introduction form for $\forall^\nu X.A$ is the same as the ordinary universal quantifier. On the other hand, the elimination form is different, because it always instantiates the polymorphic function with a freshly generated type. I read this elimination form as saying, given a $t : \forall^\nu X.A$, one can instantiate it with a freshly created type Y that is defined to be isomorphic to B and bind the result to the variable x in the continuation u. The side condition that $\Gamma \vdash C$ where $u : C$ ensures that this freshly created type definition doesn't leak out of scope. There are two versions of this rule: one for when t is known to have type $\forall^\nu X.A$ and another for when it is dynamic, similar to the treatment of function types in Siek and Taha [2006].

I give semantics for this language extension using the open sum type in Figure 13. First, I define the interpretation of contexts Γ. Term variables are interpreted as usual. I interpret abstract type variables X in Γ as pairs of a type interpretation X and a term variable for a case c_X : Case X. Next, known

[3]Something like Typed Racket's occurrence typing or some form of dependent typing could manage this.
[4]Note that this extensibility can also be used to model creation of nominal types

$$
\begin{aligned}
[\![\cdot]\!] &= \cdot \\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : |A|_\Gamma \\
[\![\Gamma, X]\!] &= [\![\Gamma]\!], X, c_X : \text{Case } X \\
[\![\Gamma, X \cong A]\!] &= [\![\Gamma]\!], c_X : \text{Case } |A|_\Gamma \\
|?|_\Gamma &= \mu D.\texttt{bool} + (D \times D) + (D \to D) + (\forall X.\text{Case } X \to D) + \text{OSum} \\
|X|_\Gamma &= X && (X \in \Gamma) \\
|X|_\Gamma &= |A|_\Gamma && (X \cong A \in \Gamma) \\
e_{X;\Gamma} &= \lambda x : |X|_\Gamma.\texttt{inj}_{c_X}\ x \\
p_{X;\Gamma} &= \lambda d : |?|_\Gamma.\texttt{match}\ d\ \texttt{with}\ c_X\{\texttt{inj}\ x.x \mid \mho\} \\
|\forall^\nu X.A|_\Gamma &= \forall X.\text{Case } X \to |A|_\Gamma \\
e_{\forall^\nu X.A;\Gamma} &= \lambda f : \forall X.\text{Case } X \to |A|_\Gamma.\text{in}_\forall\ (\Lambda X.\lambda c_X : \text{Case } X.e_{A;\Gamma,X}(f[X]c_X)) \\
e_{\forall^\nu X.A;\Gamma} &= \lambda d : |?|.\texttt{case}\ d\,\{\text{in}_\forall\ f.\Lambda X.\lambda c_X : \text{Case } X.p_{A;\Gamma,X} \mid \texttt{else}\ \mho\} \\
[\![\texttt{seal}_X t]\!]_\Gamma &= p_{A;\Gamma}(e_{B;\Gamma}[\![t]\!]) && (X \cong A \in \Gamma, \Gamma \vdash t : B) \\
[\![\texttt{unseal}_X t]\!]_\Gamma &= p_{X;\Gamma}(e_{B;\Gamma}[\![t]\!]) && (X \cong A \in \Gamma, \Gamma \vdash t : B) \\
[\![\Lambda X.t]\!] &= \Lambda X.\lambda c_X : \text{Case } X.[\![t]\!] \\
[\![\texttt{let}\ x = t\,\{Y \cong B\}\texttt{in}\ u]\!] &= \texttt{let}\ f = [\![t]\!]; \texttt{newcase}_{|B|}\ c_Y; \texttt{let}\ x = f[|B|]c_Y; [\![u]\!] \\
&&& (\text{when } t : \forall^\nu X.A \text{ for some } A) \\
[\![\texttt{let}\ x = t\,\{Y \cong B\}\texttt{in}\ u]\!] &= \texttt{let}\ f = p_{\forall^\nu X.?}[\![t]\!]; \texttt{newcase}_{|B|}\ c_Y; \texttt{let}\ x = f[|B|]c_Y; [\![u]\!] && (\text{when } t : ?)
\end{aligned}
$$

Figure 13: Interpreting Sealing and Fresh Polymorphism

type variables $X \cong A$ are interpreted as cases $c_X : \text{Case } A$. The dynamic type is given the static type interpretation described above.

Next, I interpret type variables. If X is an abstract type variable, it is interpreted as a type variable, but if it is known $X \cong A$, then it is interpreted the same as A. In either case, the embedding-projection pair associated to X is constructed using the case c_X. Polymorphic functions are interpreted as noted before, and their embedding-projection pair is similar to that of function types. I use $\text{in}_\forall$ as an abbreviation for the actual sequence of injections that tag $\forall^\nu$ types in our interpretation of ?, which is: `inr inr inr inl` .

Finally I present the term translation. The $\texttt{seal}_X$ and $\texttt{unseal}_X$ forms are just interpreted as insertion of the casts associated to the side-condition $A' \lesssim A$ and $A' \lesssim X$ from the typing-rule, since the static type interpretation of X is actually the same as A. Next, I interpret polymorphic functions to bring both a type variable X and a case variable c_X into scope. Finally, for type instantiation, I allocate a new case and pass that to the polymorphic function along with the type.

Forcing polymorphic function application to be written in this form is essentially the same as an ANF-restriction on type application. For instance, applying the identity function $t : \forall^\nu X.X \to X$ to an argument $u : A$ would be written as

$$\texttt{let}\ f = t\,\{Y \cong A\}\texttt{in}\ \texttt{unseal}_Y(f(\texttt{seal}_Y u))$$

However, I conjecture that all System F terms can be embedded in this extended language, and that parametric reasoning is preserved. Making this precise is difficult, though, because the additional power of having a universal type is known to violate full abstraction, so a more refined definition of "parametric reasoning" is necessary [Devriese et al., 2017].

In a draft paper [New et al., 2019a], we greatly expand on this language, using a syntax without an ANF restriction, but where the binding of the newly created type is propagated outwards. We also include existential types, and give an operational semantics for a cast calculus similar to the surface language and prove a simulation theorem for the translation into a typed target language. Finally, we prove parametricity and graduality theorems for the calculus, which follow by essentially the same logical relations method of previous work. One novelty of this logical relation is that it is general enough to prove both the graduality theorem and the parametricity theorem: the parametricity theorem is just the reflexivity case of graduality.

Chapter	Deadline
EP Pair Semantics	New and Ahmed [2018]
Parametricity & Graduality	Submitted New et al. [2019a] Revisions October 2019
Gradual Type Theory	New and Licata [2018], New et al. [2019b] CBV Version, weakening η discussion December 2019
Remaining writing	January-February 2020
Submit Thesis	late February 2020
Defense	April 2020

Figure 14: Work Plan

7 Remaining Work

The plan for the remaining work of the dissertation is presented in Figure 14[5]. Most of the work of the thesis has already been included in a conference publication [New and Licata, 2018, New and Ahmed, 2018, New et al., 2019b], specifically chapters based on the ep pair semantics (Section 4) and gradual type theory (Section 5). The exception is the final core chapter on our gradual polymorphic language (Section 6), which is currently under submission [New et al., 2019a]. The remaining technical work is mostly minor, but will include providing a call-by-value variant of gradual type theory that elaborates into call-by-push-value gradual type theory. I will also expand discussion of how to apply our framework to languages with procedures that don't satisfy even the call-by-value function η law, such as those that include features like pointer equality, inspection of closures, etc. Finally, I will flesh out how to translate System F into our fresh polymorphic language and in what sense parametric reasoning is preserved by this embedding.

References

Amal Ahmed, Robert Bruce Findler, Jeremy Siek, and Philip Wadler. Blame for all. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 201–214, January 2011.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom*, 2017.

Matteo Cimini and Jeremy G. Siek. 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, 2016.

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

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

Dominique Devriese, Marco Patrignani, and Frank Piessens. Parametricity versus the universal type. *Proc. ACM Program. Lang.*, 2(POPL):38:1–38:23, December 2017. ISSN 2475-1421. doi: 10.1145/3158126. URL `http://doi.acm.org/10.1145/3158126`.

Robby Findler and Matthias Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, April 2006.

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

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

[5]I will also be applying for faculty positions this year, so the length of time for writing is quite conservative to accommodate for submitting applications and interviewing. I will also be getting married in late March

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, 2010.

Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. In *International Conference on Functional Programming (ICFP), St. Louis, Missouri*, 2018.

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

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom*, 2017.

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

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

Max S. New and Daniel R. Licata. Call-by-name gradual type theory. *FSCD*, 2018.

Max S. New, Dustin Jamner, and Amal Ahmed. Graduality and parametricity: Together again for the first time, 2019a. URL `http://maxsnew.github.io/docs/poly.pdf`.

Max S. New, Daniel R. Licata, and Amal Ahmed. Gradual type theory. *Proc. ACM Program. Lang.*, 3(POPL):15:1–15:31, January 2019b. ISSN 2475-1421. doi: 10.1145/3290328. URL `http://doi.acm.org/10.1145/3290328`.

John C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

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

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

Jeremy Siek, Michael Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages*, SNAPL 2015, 2015.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (Scheme)*, pages 81–92, September 2006.

Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *ACM Symposium on Principles of Programming Languages (POPL), Venice, Italy*, 2004.

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

Matías Toro, Ronald Garcia, and Éric Tanter. Type-driven gradual security with references. *ACM Transactions on Programming Languages and Systems*, 40(4), December 2018. URL `http://doi.acm.org/10.1145/3229061`.

Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *Proc. ACM Program. Lang.*, 3(POPL):17:1–17:30, January 2019. ISSN 2475-1421. doi: 10.1145/3290330. URL `http://doi.acm.org/10.1145/3290330`.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, 2017.

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