Reëxamining the Gradual Guarantee

ANONYMOUS AUTHOR(S)

Boyland introduced the *Gradual Guarantee* (further refined by Siek et al.) and observed that structural type matching violates it, using the pattern-matching system of the Grace language. In this paper we show that there is an appropriate semantics for structural typecase in a gradually-typed language which satisfies the guarantee on Boyland's example, and formalize that behaviour. We then show that, even with this semantics, and in fact even without either structural typing or typecase, the Gradual Guarantee is unsatisfied by virtually all practical gradually-typed languages, including one developed by the authors of the refined criteria for gradual typing. We propose a new Gradual Guarantee that is achievable, and an alternative criterion that renders the original guarantee satisfiable at the expense of drastically limiting the range of languages that can be gradual.

ACM Reference format:
Anonymous Author(s). 2017. Reëxamining the Gradual Guarantee. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 29 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Gradual typing is a popular paradigm for languages developed today, offering the promise of getting the best of both worlds from dynamically- and statically-typed languages. In a gradually-typed language typed and untyped code can be used together, and partially-typed programs can execute until a type error occurs at run-time.

The increasing number of supposed gradual languages led to a range of subtly different interpretations of gradual typing, which led Siek et al. (2015) to introduce formal Refined Criteria for Gradual Typing. One of the refined criteria is the *Gradual Guarantee*, first introduced by Boyland (2014), which states that the presence or absence of type annotations should not cause a well-typed program to have different behaviour.

Both Boyland (2014) and Siek et al. (2015) suggest dynamic structural type tests as a problem for the guarantee, and that a language with these features cannot meet the criteria to be considered gradual. In this paper we show that neither structural typing nor type tests are fundamentally incompatible with the guarantee, and propose a new mechanism for run-time structural tests that conforms to the requirements of the guarantee. We present a formal model of the matching semantics and a novel system of casts using the object as ground truth to support type safety, and show that under our proposed mechanism structural type tests cannot affect program behaviour.

We go on to show that, nonetheless, virtually all supposedly gradual language implementations are **not** gradual according to the Refined Criteria and Gradual Guarantee. While the formal models of gradual typing may satisfy the guarantee, real-world implementations (including one by some of the authors of the refined criteria) make compromises that allow type annotations to affect the evaluation of the program.

As a result, we propose a new Gradual Guarantee that real-world languages can achieve. As an alternative, we also propose an additional criterion that renders the existing Gradual Guarantee satisfiable while restricting the range of gradual languages significantly.

The next section introduces necessary terminology and background, and explains the origin and role of the Gradual Guarantee. Section 3 presents our new structural matching mechanic and shows that it resolves the motivating case of the Gradual Guarantee. Section 4 gives a formal representation of our structural matching semantic, and a system of casts that preserve gradual type safety under it. Section 5 illustrates that the Gradual

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

Guarantee does not hold in real languages, and Section 6 proposes an updated guarantee which does. Section 7 positions this work amongst related work, and Section 8 concludes.

1.1 Contributions

The contributions of this paper are:

- The definition of a suitable semantics for dynamic structural type tests in a gradually-typed language.
- A formal model of casts using object inspection implementing that semantics.
- An updated Gradual Guarantee addressing inconsistencies we have identified.

2 BACKGROUND

Gradual typing allows programs to be only partially typed and still execute, permitting a program to begin its life as dynamically-typed and have type annotations added gradually over time, as well as permitting different parts of the program to adopt different typing approaches permanently. The type annotations do not affect the dynamic behaviour of the program, only providing additional error checking. Gradual type systems include a distinguished "unknown" or "dynamic" type, which is statically compatible with any type. Run-time checks are required to preserve type safety when static and dynamic code meet.

Structural typing for objects represents types as sets of methods, where an object belongs to a type if it contains methods corresponding to each method in the type. A structural type says nothing about the implementation of the object, or what other methods it may contain. Structural types permit ad-hoc types to be created after the fact without the object's knowledge. Structural typing can be contrasted with *nominal* typing, where types and subtypes are determined by labels placed on the object, often from a defining class.

Type tests are run-time inspections of objects to establish whether they belong to a certain type or not. Java's instanceof, C♯'s is, and Smalltalk's isKindOf: are mechanisms for type tests. These tests are often frowned upon as poor object-orientation, but some structured "*typecase*" mechanisms embrace type testing as part of hybrid programming paradigms, bundling multiple type tests together and branching for each case.

Boyland (2014) reported a problem of structural type tests in a gradually-typed language. Boyland was working with the pattern-matching system (Homer et al. 2012) of the structurally- and gradually-typed Grace language (Bruce et al. 2013), and their formalism Tinygrace (Jones and Noble 2014). The issue identified is that, when performing a type test on a structural type, type annotations on parameter and return types affect whether the match succeeds or not. The following example adapted from Boyland (2014) illustrates the issue:

```
type Image = { draw(c : Canvas) → Object }
type Cowboy = { draw(n : Nat) → Cowboy }
var o := object { method draw(n : Nat) → Cowboy { self } }
match(o)
  case { a : Image → 1 }
  case { b : Cowboy → 2 }
  case { _ : Object → 3 }
```

The two types, Image and Cowboy, both have a single method called draw. The methods have different parameter and return types, so an Image is not a Cowboy, and a Cowboy is not an Image. The match case... construct at the bottom inspects an object, o, which has a draw method corresponding to the one in Cowboy.

The match will test the target object against each possible case in turn, and evaluate to the result of the first branch with a matching type. Checking the object against the Image type will fail, because Canvas is not a supertype of Nat. Checking against Cowboy will succeed, and the match evaluates to 2.

If the type annotations within o are erased, leaving:

```
var o := object { method draw(n : Unknown) → Unknown { self } }
```

and the same match performed, a different outcome occurs. When testing whether o belongs to the Image type, the check will succeed because the dynamic type Unknown is compatible with both Canvas and Object. Evaluation will proceed down the first branch, resulting in 1. The presence or absence of type annotations thus affects the semantics of the program. This situation led Boyland to formulate the *Gradual Guarantee*:

> If an expression e_1 evaluates without error in one step to e_2, then any expression e_1' with fewer annotations ($e_1 \prec e_1'$) also evaluates in zero or more steps to e_2' where $e_2 \prec e_2'$.

Informally, the guarantee states that a program that evaluates to a given result will evaluate to the same result if any number of type annotations is removed. Boyland holds this guarantee as a necessity of gradual typing. The Gradual Guarantee was subsequently incorporated into Siek et al. (2015)'s refined criteria for gradual typing, which sought to put the thitherto informal notions of what it meant for a language to be gradual onto firmer footing. The modern notation $e_1 \sqsubseteq e_2$ is now used to represent that the expressions e_1 and e_2 are equal, except that for all of the type annotations in e_1, the type is the same or less precise than the corresponding type in e_2.

Boyland (2014) and Siek et al. (2015) both refer to this issue as the result of dynamic type tests under structural typing. The implication in both cases is that without structural types, or without type tests, the behaviour of a program will be unaffected by type annotations. We show that structural types and type testing, separately or together, are neither necessary (Section 5) nor sufficient (Section 3) to violate the Gradual Guarantee.

3 SHALLOW STRUCTURAL TYPE MATCHING

The issue raised by Boyland (2014) is that removing a parameter- or return-type annotation from an object expanded the set of types that matched the object, potentially changing the behaviour of the program when used in a typecase. The Tinygrace calculus of Jones and Noble (2014) used by Boyland treats typecase matching in this way — reusing the structural subtyping used for static checking in the dynamic semantics. Fundamentally, type annotations affect the dynamic behaviour of the program in this case.

Tinygrace's mechanism is not the only approach available, however. In fact, all current implementations of the Grace language itself implement a different behaviour: Minigrace (Bruce et al. 2013) considers a type to match if methods by the correct *names* appear within the object, without regard for type annotations in the method signature, while Kernan (Homer 2016) and Hopper (Jones 2015) also incorporate method arity.

These behaviours were initially considered as incomplete by the authors[1], with an intention of fuller structural checks being implemented in future. We propose that a behaviour very similar to this is in fact desirable and the correct approach to a dynamic structural check.

Specifically, we propose that **a (runtime) object should be considered to be matched by a structural type if and only if, for every method in the type, all valid calls of that method with the object as receiver would be accepted by the object up to mismatches in the types of parameters and arguments**. In the simple case of Java-style methods, this means that for every method in the type a correspondingly-named method exists in the object that accepts the same *number* of arguments, regardless of type.

That is, if all parameter and return types were erased from the program, would every method call on the object that is valid on the type cause a method to begin executing? If so, the match should succeed. Otherwise, at least one method exists in the type which has no hope of being called on the object, and the match must fail.

While it seems counterintuitive, ignoring type annotations when matching structural types is the correct thing to do. In a gradually-typed language, a dynamic check in this fashion is inherently a best-guess: a method declared as returning the unknown type may or may not ultimately give a result consistent with the expected type, and there is no (general) way to know until it is called. Even then, it may do something else the next time: the assumption can be falsified, but never confirmed. On the other end, the *type* may include the unknown type as a parameter type: a method requiring a string is (already) consistent with this type, but valid calls on the

[1] Personal communication with the authors of each system.

type can cause errors. In a gradual language, which already embraces this uncertainty, it is natural to follow the same principle, and to use the same mechanisms to handle the situation when the assumption is wrong.

When the annotations are ignored, a consistent behaviour emerges. What we can know for certain is which methods exist, and that assumption will not be invalidated. This consistency allows upholding the gradual guarantee on these programs.

3.1 Shallow matching and the Gradual Guarantee

Recall Boyland (2014)'s example from Section 2. Erasing the parameter and return types from the object o caused the match…case to have a different result, as the Image type then matched before Cowboy. As removing the type annotation caused the result of the program to change, the gradual guarantee was violated.

With our new matching semantic, the guarantee is upheld for this program:

- Without annotations, the first case, Image, will be matched and the result will be 1 (as before).
- In the fully-annotated program, o has a method draw accepting a single argument, as required by Image, and so the Image case still matches. The result will again be 1.

In fact, no removal of annotations on either object or type can ever cause the match to proceed down a different branch. When this semantic is used, matching and structural typing do not inhibit the gradual guarantee. Section 4 formalizes this behaviour, and proves that removing annotations cannot affect the result.

3.2 Type safety

It appears that this semantic is not type safe. Strictly, the match operation is separable from a type-cast: matching a type against an object provides information on what methods are present, but nothing is known about their inputs and outputs, so methods with parameters cannot be requested. Type safety is thus not immediately affected by matching in this way.

It is usual, however, that the type-case is accompanied by a type-cast of the reference to the target type as well. In a gradually-typed language, this is not a problem. Potentially ill-founded type-casts are standard occurrences whenever a dynamically-typed value is passed to a statically-typed function or method, and all of the same mechanisms continue to apply.

Vitousek et al. (2014) describe three different semantics available in implementing sound gradual typing, applying when an untyped value is used in a typed context. In the *Guarded* semantic, objects are wrapped (chaperoned) to enforce the stated type on inputs and outputs. Each method has a wrapper method that checks that the arguments and return value meet the required type. Under the *Transient* semantic, every use of an object is checked against its locally-declared type. Return values from methods are checked on the client side. With the *Monotonic* semantic, the object is modified whenever it passes a check to remember and enforce all its contracts. The object enforces that the type requirements are met within itself.

All three of these are equally applicable to this newly-cast reference as they are to method arguments or returned values. In fact, a sensible implementation of such a system would be to use first-class functions for each branch, directly using the existing gradual functionality of parameters. This parallel continues further, which we discuss in Section 5.

3.3 Extensions

The important aspect of our proposal is that any untyped call that is valid on the type should be valid on the concrete object. This property might be termed *typeless consistency*: with parameter and return types disregarded, will the object accept all calls permitted by the type? Languages support a range of different calling conventions, and while we are focusing only on methods with a single ordered list of parameters here, extensions preserving the overall goal should be possible.

As the simplest example, in a language with variadic parameters, a method with a variadic parameter accepts any number of arguments greater than or equal to the position of the variadic argument. In this case, a method with a minimum argument count of n will be consistent with any method in a type with n *or more* parameters.

Similarly, in a language with a more complex calling convention, such as keyword arguments, a method in an object should be compatible with the signature in a type provided that any valid call of the method in the type is accepted by the object up to parameter type errors. If the method in the type includes additional keyword arguments that are simply ignored by the method in the object, or are coalesced with positional parameters, then the calls remain valid and the object is typelessly consistent with the type.

When using an object model that includes handling for failed method calls, such as Smalltalk's doesNotUnderstand:, it is conceivable that *any* method is accepted by an object with such a handler. In this case it is not perfectly obvious whether the object should be matched by all types, or whether only "real" declared methods should be incorporated. We suggest the latter course as a default, as the practical result of the other approach is that such objects cannot usefully be used with type-matching at all.

3.4 Non-structural type tests

Rather than structural type tests, it is in fact *higher-order type tests* that are a problem. Even in a nominally-typed system, matching function types, generic parameters, or collection types can create violations of the Gradual Guarantee. In this light, a structural type can be seen as merely a map from method names to function types.

Testing whether a first-class function meets a particular function type, for example, should follow the same approach as we have proposed for structural types: optimistically matching if the arity is valid, while enforcing soundness dynamically. Otherwise, the type annotations on the function declaration affect the result of the program exactly as the parameter type declarations on methods affected the structural checks in Tinygrace.

Shallow nominal tests, of the "is this an instance of Animal?" variety, are compatible with the gradual guarantee. Tests that bring other types into the equation are not. It is depth, not typing discipline, that matters.

4 A FORMAL MODEL OF MATCHING AND CASTS

We define Mold, a blameless cast calculus with a matching construct similar to the Tinygrace system (Jones and Noble 2014) used by Boyland (2014), with a structural *match* construct obeying the semantics from Section 3, and a system of casts and coercions that preserve type safety up to invalidated casts. Mold is a potential target for a cast-insertion procedure on a gradual language that also replaces dynamic types with $\top$. We are more interested in the semantics of the casts and matching than the translation from a corresponding gradual language, so we only include the cast calculus here.

4.1 Syntax

The grammar of Mold is defined in Figure 1. Types are separated into a mutually recursive hierarchy of union types T, structural types S, and method definition types D. We write the empty union type as $\bot$ and a union containing a single, empty structural type as $\top$.

We follow DOT (Rompf and Amin 2016) in using z as a typical 'abstract' variable and y for a 'concrete' store reference, with x ranging over both. An object expression $\{\, z \Rightarrow \overline{d} \,\}$ constructs an object with self variable z and method definitions $\overline{d}$. Any term can be raised with $\Uparrow t$, and any raise can be rescued using $t_1 \,\overline{\Uparrow}\, \{\, z \to t_2 \,\}$, where t_2 is evaluated if t_1 raises with z bound to the value in the raise. A match $t_1 \ni m \{\, z \to t_2 \,\} \{\, z \to t_3 \,\}$ reflects on the value of t_1 and branches into t_2 if a method named m is present, or t_3 if not, with z bound to the value of t_1.

The unusual number of evaluation contexts is the result of including both exception handling and early merging of casts. The context E is the typical context for congruence evaluation, whereas F excludes terms behind a rescue and is used to avoid raising a value beyond the nearest surrounding rescue. The context G is like F except

Grammar

$$m, x, y, z \in \textsc{Var}, \quad T \in \textsc{Type}, \quad S \in \textsc{Struct}, \quad D \in \textsc{Decl},$$
$$d \in \textsc{Def}, \quad t \in \textsc{Term}, \quad v \in \textsc{Value}, \quad \Gamma \in \textsc{Env}, \quad \sigma \in \textsc{Var} \rightharpoonup \wp(\textsc{Def})$$

$$T ::= \overline{S} \qquad \textit{(Union type)}$$
$$S ::= \{\, \overline{D} \,\} \qquad \textit{(Structural type)}$$
$$D ::= m : T \to T \qquad \textit{(Definition type)}$$
$$d ::= m(z : T) : T = t \qquad \textit{(Definition)}$$
$$t ::= x \mid \{\, z \Rightarrow \overline{d} \,\} \mid t.m(t) \mid \Uparrow t \mid t \Uparrow \{\, z \to t \,\} \mid t \ni m\,\{\, z \to t \,\}\,\{\, z \to t \,\} \mid t : S \qquad \textit{(Term)}$$
$$v ::= y \mid y : S \qquad \textit{(Value)}$$

Environments

$$\Gamma ::= \cdot \mid \Gamma, z : T \qquad \textit{(Typing environment)}$$

Evaluation contexts

$$E ::= F \mid F[E \Uparrow \{\, z \to t \,\}] \qquad \textit{(Term context)}$$
$$F ::= \square \mid G \qquad \textit{(Rescue-free context)}$$
$$G ::= F.m(t) \mid v.m(F) \mid \Uparrow F \mid F \ni m\,\{\, z \to t_1 \,\}\,\{\, z \to t_2 \,\} \mid \square : S \mid I[F] : S \qquad \textit{(Sub-term context)}$$
$$H ::= \square \mid E[I] \mid E[\square \Uparrow \{\, z \to t \,\}] \qquad \textit{(Direct cast-free context)}$$
$$I ::= \square.m(t) \mid v.m(\square) \mid \Uparrow \square \mid \square \ni m\,\{\, z \to t_1 \,\}\,\{\, z \to t_2 \,\} \qquad \textit{(Immediate sub-term context)}$$

Fig. 1. Mold Grammar

it only matches a sub-term. All three of these contexts cannot match a sub-term that appears anywhere inside of two directly nested casts, since these casts need to be merged before any evaluation in their body can occur.

The context H is like E except it cannot include a hole that is directly surrounded by a cast, to ensure that casts are merged outside-in and that no more than one hole is available when attempting to merge a succession of three or more casts. The context I is internal to the grammar, used to avoid duplication in defining G and H.

4.2 Matching

Because a match in Mold only looks for a matching method name and not a full signature, it does not tell you anything about the type annotations on the method in the case that it does exist, only that such a method is present. The behaviour of a match distinguishes this match form from Tinygrace's (Jones and Noble 2014), which matched on a full type.

The match construct allows for the encoding of pattern matching against a structural type as described in Homer et al. (2012). The match method in the reified object of the type $\{\, m_1 : T_1 \to T_2, m_2 : T_3 \to T_4 \,\}$ can be encoded in Mold as:

```
method match(a : Object) → MatchResult {
    a ∋ m1 { b → b ∋ m2 { c → success(c) } { c → failure(c) } } { b → failure(b) }
}
```

The matching only works on a shallow level, identifying if methods with the appropriate name are present, so matching against a structural type only performs a shallow test. When typing the body of the match/case construct, a typing judgment can only assume that the shallow description of the pattern applies to the object.

4.3 Casts and coercions

As a cast calculus, Mold includes syntax for expressing assumptions about the type of an object. The existence of a method can be assumed with a *coercion*, which ensures that a failed assumption safely results in a raise instead of the evaluation getting stuck. Higher-order *casts* then chaperone objects to maintain the remaining unchecked assumptions about the methods, expanding to a coercion when it becomes possible to check an assumption. Casts allow the 'then' branch of a match to check for the presence of a method, then make assumptions about the inputs and outputs of that method.

Mold's matching form examines the structure of an object; this can be used to discover whether a method is present in the object at run-time. Invalidated assumptions are encoded using the raise form.

We use the term 'coercion' to describe an examination of an object's structure with match expressions in an attempt to give it a more specific type that it already has. Coercing a term t to a type that assumes it has a method m can be encoded as the following match expression:

$$t \ni m\,\{\,z \to z\,\}\,\{\,z \to \Uparrow z\,\}$$

If the value of t has the relevant method, then the match will reduce directly to the value unchanged, otherwise the value is raised to indicate that the coercion has failed.

A match only examines the name of the methods in the object, not the types annotating the methods. Although the type of the coercion above contains a definition type with the name m, the match only guarantees that the method is present. Any typing judgment must assign the type $m : \bot \to \top$ to the method (unless there was already information about this method in the type of t, in which case the coercion is useless because we already knew that the method was present).

Mold also includes a syntax for casts $t : S$, indicating the term t is assumed to have the type expressed by the structural type S. Casts encode the higher-order assumptions about an object's type, chaperoning a reference to the object and applying the relevant coercion when a method is called that has assumptions that must be upheld.

Assuming that a method m in a term t accepts a Number and returns a Boolean is written:

$$t : \{\, m : \mathsf{Number} \to \mathsf{Boolean}\,\}$$

The cast only makes assumptions about the type annotations on the methods, and does *not* assume the existence of the method m. The term t must be known to contain a method m for the cast to be well-typed.

By combining the coercion of the existing match form and the new casts, any structural assumption can be applied to a term. The combination of the coercion and cast above is:

$$t \ni m\,\{\,z \to z : \{\, m : \mathsf{Number} \to \mathsf{Boolean}\,\}\,\}\,\{\,z \to \Uparrow z\,\}$$

After evaluating t, the match inspects the resulting value to see if it contains a method m, and attaches the higher-order assumptions about the inputs and outputs if the method is present. The cast only chaperones the reference of the object that is the body of the cast: other references to the object may exist that are not constrained by the assumptions in the cast.

4.4 Generating Coercions

Coercions are generated whenever a method is requested on a cast that makes assumptions about that method. The coercion generation metafunction coerce is defined in Figure 2. The metafunction is overloaded to accumulate information at the different points in the hierarchy of a type, but only the top-most definition is explicitly used in the semantics that follow. An optimization of the application of the coerce function is defined as param,

Coercion generation

$$\begin{aligned}
&\text{coerce} : \textsc{Term} \times \textsc{Type} \to \textsc{Term}\\
&\text{coerce}(t, T) \;= \text{coerce}(t, \bot, T)\\
&\\
&\text{coerce} : \textsc{Term} \times \textsc{Type} \times \textsc{Type} \to \textsc{Term}\\
&\text{coerce}(t, T, \bot) \;= \text{cast}(t, T)\\
&\text{coerce}(t, T, (S, \overline{S_i})) \;= \text{coerce}(t, T, S, \overline{S_i}, S)\\
&\\
&\text{coerce} : \textsc{Term} \times \textsc{Type} \times \textsc{Struct} \times \textsc{Type} \times \textsc{Struct} \to \textsc{Term}\\
&\text{coerce}(t, T_1, S, T_2, \{\}) \;= \text{coerce}(t, T_1 \cup S, T_2)\\
&\text{coerce}(t, T_1, S, T_2, \{\, D, \overline{D_i}\,\}) \;= \text{coerce}(t, T_1, S, T_2, \{\,\overline{D_i}\,\}, D)\\
&\\
&\text{coerce} : \textsc{Term} \times \textsc{Type} \times \textsc{Struct} \times \textsc{Type} \times \textsc{Struct} \times \textsc{Decl} \to \textsc{Term}\\
&\text{coerce}(t, T_1, S_1, T_2, S_2, D) = \; t \ni \text{identify}(D)\,\{\, z \to \text{coerce}(z, T_1, S_1, T_2, S_2)\,\}\,\{\, z \to \text{coerce}(z, T_1, T_2)\,\}
\end{aligned}$$

Cast generation

$$\begin{aligned}
&\text{cast} : \textsc{Term} \times \textsc{Type} \to \textsc{Term}\\
&\text{cast}(t, \bot) \;= \Uparrow t\\
&\text{cast}(t, \top) \;= t\\
&\text{cast}(t, T) \;= t : \text{shallow}(T)\\
&\\
&\text{shallow} : \textsc{Type} \rightharpoonup \textsc{Struct}\\
&\text{shallow}(S) \;= S\\
&\text{shallow}((S, \overline{S_i})) \;= \text{shallow}(S, \text{shallow}(\overline{S_i}))\\
&\\
&\text{shallow} : \textsc{Struct} \times \textsc{Struct} \to \textsc{Struct}\\
&\text{shallow}(\{\, D_1, \overline{D_i}\,\},\ \{\,\overline{D_j}, D_2, \overline{D_k}\,\}) \;= \{\, \text{shallow}(D_1, D_2)\,\} \cap \text{shallow}(\{\,\overline{D_i}\,\}, \{\,\overline{D_j}, \overline{D_k}\,\})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where } \text{identify}(D_1) = \text{identify}(D_2)\\
&\text{shallow}(\{\, D, \overline{D_i}\,\},\quad S) \;= \{\, D\,\} \cap \text{shallow}(\{\,\overline{D_i}\,\}, S)\\
&\text{shallow}(\{\},\quad S) \;= S\\
&\\
&\text{shallow} : \textsc{Decl} \times \textsc{Decl} \rightharpoonup \textsc{Decl}\\
&\text{shallow}(m : T_1 \to T_2, m : T_3 \to T_4) \;= m : T_1 \cap T_3 \to T_2 \cup T_4
\end{aligned}$$

Parameter optimization

$$\begin{aligned}
&\text{param} : \textsc{Term} \times \textsc{Type} \times \textsc{Type} \to \textsc{Term}\\
&\text{param}(t, T_1, \bot) \;= t\\
&\text{param}(t, T_1, T_2) \;= \text{coerce}(t, T_1)
\end{aligned}$$

Fig. 2. Coercion generation metafunctions

which avoids applying a coercion to an argument if no assumptions were made about the type of the corresponding method parameter.

A coercion generated by a request checks that the arguments satisfy the type annotations of the parameters on the method, and the result of the request is coerced to the return type in the cast. Consider a request on a cast that assumes that a method accepts anything and returns an object with some method named a:

$$(y : \{\, m : \top \to \{\, a : \bot \to \top\,\}\,\}).m(t)$$

The object at y contains a method m that requires its input to contain a method b:

$$\sigma(y) = m(z : \{\, b : \bot \to \top \,}) : \top = \cdots$$

Requesting m on the cast removes the cast, but wraps the input in a coercion that tests for the presence of b, and wraps the remaining request in a coercion that tests for a's presence. The request unfolds the cast to become:

$$\text{coerce}(y.m(\text{param}(t, \{\, b : \bot \to \top \,}, \top)), \{\, a : \bot \to \top \,})$$

Each of the coercions becomes a match expression around their inputs.

$$y.m(t \ni b\, \{\, z \to z \,}\, \{\, z \to \Uparrow z \,}) \ni a\, \{\, z \to z \,}\, \{\, z \to \Uparrow z \,}$$

If the input does not match the type expected by the method, or the method returns a value that does not satisfy the assumption of the cast, then an error is raised.

There appears to be an interesting asymmetry in the cast expansion: while the result of the request is checked against the return type in the cast, the arguments of the request are checked against the parameter types of the method that actually appears in the object that is the receiver of the request, and the parameter types in the cast are ignored. An assumption about the return type of a method adds information to the expected type of the request, but an assumption on parameter types *forgets* information about the expected type of an input to the method. Without also including a source type with the target type in the cast, the information about what was originally expected as the type of an input is lost when a cast is applied.

Using the parameter types in the method rather than in the cast is part of a broader philosophy of treating the object that is the body of the cast as the ultimate truth about that object's type. The higher-order coercions of Henglein (1994) and casts of Siek and Taha (2006) and Wadler and Findler (2009) only ever consider the consistency of the source and target of a cast, never the actual type of the underlying body. The advantage of their approach is that type annotations outside of a cast are irrelevant to the execution of a program and can be erased before run-time, whereas all parameter type annotations must be preserved at run-time in our implementation (though this is already true of Grace programs, because they are translated into contracts at run-time). The disadvantage of not examining the body of the cast when attempting to determine if the assumptions of a cast are upheld is that a cast that makes valid assumptions may still fail: we discuss this problem in Section 4.7.

The apparent asymmetry is resolved by considering that the parameter types in the cast are not wholly useless: they are still used by the type system to type check the inputs before run-time. The parameter types in the cast correspond to the return type on the *method that is ultimately called*, not the return type in the cast: in both cases we trust the type system to check the relevant terms under the assumption that the surrounding context is sensible. The reverse of this correspondence is that the return type in the cast corresponds to the parameters on the method, in that they both must be checked at run-time.

Retaining both the source and target of a cast — and reversing these types when distributing a cast over an input — instead of investigating the parameters on the actual method being called would also prevent casts from fulfilling their most useful function in Mold: calling methods discovered by a match. Any method discovered by a match has inputs of type $\bot$ within the 'then' branch of the match, but the parameter types on the actual method may be less restrictive. If the input types are checked by reversing the direction of a cast, then casting the input from $\bot$ to some type T would require checking that the input (of type T) satisfies $\bot$, which would mean the methods are still unable to be called (since $\bot$ is empty).

Consider the following use of a match, checking if a method m appears in the value of a term t, and requesting it with some input x if it does.

$$t \ni m\, \{\, z \to z.m(x) \,}\, \cdots$$

Either this match is not well-typed, or the input x has type $\bot$. In the latter case the request to m can never be evaluated because the input has no value: the argument will never reduce to a value, either diverging or resulting in a raise.

The application of a cast to the receiver z can be used to make the request well-typed, by assuming that the m method will actually accept x (of some type T):

$\boxed{\sigma \mid t \longrightarrow \sigma \mid t}$

$$\text{(E-Obj)}\ \frac{y \text{ fresh} \qquad \overline{\text{identify}(d)} \text{ unique}}{\sigma \mid \{\, z \Rightarrow \overline{d} \,\} \longrightarrow \sigma(y \mapsto [y/z]\overline{d}) \mid y} \qquad \text{(E-Sfe)}\ \frac{}{\sigma \mid v \Uparrow \{\, z \to t \,\} \longrightarrow \sigma \mid v} \qquad \text{(E-Req)}\ \frac{m(z : T_1) : T_2 = t \in \sigma(y)}{\sigma \mid y.m(v) \longrightarrow \sigma \mid [v/z]t}$$

$$\text{(E-Cst)}\ \frac{S(m) : T_1 \to T_2 \qquad m(z : T_3) : T_4 = t \in \sigma(y)}{\sigma \mid (y : S).m(v) \longrightarrow \sigma \mid \text{coerce}(y.m(\text{param}(v, T_3, T_1)), T_2)} \qquad \text{(E-Rsc)}\ \frac{}{\sigma \mid F[\Uparrow v] \Uparrow \{\, z \to t \,\} \longrightarrow \sigma \mid [v/z]t}$$

$$\text{(E-Fst)}\ \frac{\overline{d} = \sigma(v) \qquad m \in \overline{\text{identify}(d)}}{\sigma \mid v \ni m\,\{\, z \to t_1 \,\}\{\, z \to t_2 \,\} \longrightarrow \sigma \mid [v/z]t_1} \qquad \text{(E-Snd)}\ \frac{\overline{d} = \sigma(v) \qquad m \notin \overline{\text{identify}(d)}}{\sigma \mid v \ni m\,\{\, z \to t_1 \,\}\{\, z \to t_2 \,\} \longrightarrow \sigma \mid [v/z]t_2}$$

$\boxed{\sigma \mid t \longmapsto \sigma \mid t}$

$$\text{(E-Cng)}\ \frac{\sigma \mid t \longrightarrow \sigma' \mid t'}{\sigma \mid E[t] \longmapsto \sigma' \mid E[t']} \qquad \text{(E-Rse)}\ \frac{}{\sigma \mid G[\Uparrow v] \longmapsto \sigma \mid \Uparrow v} \qquad \text{(E-Mrg)}\ \frac{}{\sigma \mid H[(t : S_1) : S_2] \longmapsto \sigma \mid H[t : S_1 \cap S_2]}$$

Fig. 3. Mold reduction rules

$$t \ni m\,\{\, z \to (z : \{\, m : T \to \top \,\}).m(x) \,\}\ \cdots$$

So long as x has type T, this is now well-typed, but if the cast is required to ensure that it is correct by reversing the assumption it made about the input, the request on the cast would be transformed into the following term (where the reference y is the value of t):

$$y.m(\Uparrow x)$$

The input becomes a raise because it is a degenerate case of a coercion: there is nothing to check, because the coercion is guaranteed to immediately fail. By checking the parameter types on the actual method being called, the immediate failure of any inputs with assumptions on their types only happens if the method truly does not accept any object by including a $\bot$ parameter.

4.5 Dynamic semantics

The dynamic semantics of Mold are defined in Figure 3. Reduction is split into two relations, $\longrightarrow$ and $\longmapsto$: the former handles computation in a sub-term, and the latter uses the evaluation contexts E, G, and H to perform reductions on a larger term by either applying $\longrightarrow$ in the hole of a context with Rule E-Cng, terminating the program on an unrescued raise with Rule E-Rse, or merging the outermost casts in a term by intersecting their structural types with Rule E-Mrg.

Most of the reduction rules are straightforward. Rule E-Cst is the exception, handling reduction of requests where the receiver is a cast instead of just an object reference; this is necessary in the case that the cast makes assumptions about the type of the method being requested. In order to precisely retrieve these assumptions, a signature selection operation $S(m) : T_1 \to T_2$ is used. Signature selection retrieves the relevant signature from the structural type in the cast. The selection will still succeed if a signature with the relevant identifier is not

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

$$\text{(T-Var)}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Sub)}\ \frac{\Gamma \vdash t : T_1 \qquad T_1 <: T_2 \qquad \vdash T_2}{\Gamma \vdash t : T_2} \qquad \text{(T-Cst)}\ \frac{\Gamma \vdash t : T \qquad \vdash S \qquad T <: \mathrm{ground}(S)}{\Gamma \vdash (t : S) : T \cap S}$$

$$\text{(T-Rse)}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \Uparrow t : \bot} \qquad \text{(T-Obj)}\ \frac{\Gamma, z : \{\,\overline{D}\,\} \vdash \overline{d : D} \qquad \vdash \{\,\overline{D}\,\}}{\Gamma \vdash \{\, z \Rightarrow \overline{d}\,\} : \{\,\overline{D}\,\}} \qquad \text{(T-Req)}\ \frac{\Gamma \vdash t_1 : \{\, m : T_1 \to T_2\,\} \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1.m(t_2) : T_2}$$

$$\text{(T-Mch)}\ \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, z : T_1 \cap \{\, m : \bot \to \top\,\} \vdash t_2 : T_2 \qquad \Gamma, z : T_1 - m \vdash t_3 : T_3}{\Gamma \vdash t_1 \ni m\,\{\, z \to t_2\,\}\,\{\, z \to t_3\,\} : T_2 \cup T_3} \qquad \text{(T-Rsc)}\ \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, z : \top \vdash t_2 : T_2}{\Gamma \vdash t_1 \Uparrow \{\, z \to t_2\,\} : T_1 \cup T_2}$$

$\boxed{\Gamma \vdash d : D}$

$$\text{(T-Sig)}\ \frac{\vdash T_1, T_2 \qquad \Gamma, z : T_1 \vdash t : T_2}{\Gamma \vdash (m(z : T_1) : T_2 = t) : (m : T_1 \to T_2)}$$

$\boxed{\vdash \sigma : \Gamma}$

$$\text{(T-Sto)}\ \frac{\overline{y_i : \{\,\overline{D_{ij}}\,\}} \vdash \overline{\overline{d_{ij} : D_{ij}}} \qquad \vdash \{\,\overline{D}\,\}}{\vdash \overline{(y_i \mapsto \overline{d_{ij}})} : \overline{(y_i : \{\,\overline{D_{ij}}\,\})}}$$

Fig. 4. Mold typing judgments

present anywhere in the type, but the resulting signature will be $m : \bot \to \top$: this saves us from having to define a separate rule for handling the case where the cast does not address the method that is being requested.

Typically casts in a contravariant position such as parameter types on a method express a negative cast that needs to be reversed when flipped into a positive cast on the arguments of a request or call. If the type of a parameter is $\bot$ and this is cast to $\top$ (still a down-cast, since the polarity is reversed in a contravariant position), it would be necessary to remember that $\bot$ was the original type rather than that $\top$ is the outcome, so that reversing the parameter cast produces a new cast from the input of type $\top$ to $\bot$. Under subsumption, the parameter type on the underlying method might actually be *more permissive* than the parameter type, and casting the input to $\bot$ is not necessary to preserve the safety of the request: only satisfying the parameter on the method is required.

Casting to the real parameter types instead of just reversing the assumption allows casts to change the parameter types of a method in the 'then' branch of a match to the desired assumption, without having to cast the input to $\bot$ when the method is requested. The parameter types of any method newly discovered by a match are always $\bot$ regardless of what actually appears on the method in the store, so just reversing the assumption would leave the method unable to be requested, as its argument would always produce a raise before the request itself could be evaluated.

4.6 Static semantics

The typing judgment for Mold programs is defined in Figure 4. The subtyping used in Rule T-Sub uses standard compatibility between union types, width-subtyping between structural types, and depth-subtyping between signatures using contravariant and covariant subtyping between parameters and return types respectively.

Rule T-Mch types a match term, binding z to different types in each block body. If the method is present, then the type of z is the intersection of the original term's type and the ground structural type containing just the relevant method name: we have learned that the method is present, but know nothing about what it accepts or

what it returns. If the method is not present, then the type of z is the result of *subtracting* the method identifier from the type, removing any structural types from the union that require such a method to exist. If all of the types in the union require the method to be present, then the match is useless: the intersection adds no new information to the type, and the subtraction produces the $\bot$ type.

Rule T-Cst types a cast term, assigning the intersection of the type of the body of the cast and the structural type in the cast. The rule requires that the type of the body is a subtype of the ground of the structural type, where the ground of a structural type is the same set of signatures except all parameter types are $\bot$ and return types are $\top$. The result is that every method named in the cast must also appear in the type of the body, representing the fact that casts cannot introduce new method signatures, only refine the type annotations of existing ones.

Theorem 4.1 (Type soundness). *For any Mold store σ and term t, if $\vdash \sigma : \Gamma$ and $\Gamma \vdash t : T$, then $\sigma \mid t \longmapsto \sigma' \mid t'$, and for the resulting store and term $\vdash \sigma' : \Gamma'$ and $\Gamma' \vdash t' : T$.*

4.7 Object Inspection

Cast calculi for gradual typing conventionally use the reference type of an object before it became unknown as part of their mechanic for detecting a failed cast, while in this calculus we inspect the object itself. For typecase, it is clear that inspecting the object is required: that is a fundamental part of what it means to typecase. Inspecting the object is a necessary element to correctness *even without* typecase. We use object inspection throughout for all dynamic type checks.

Existing gradual calculi, such as that of Siek and Taha (2007), give incorrect cast errors when an object has been upcast by subsumption prior to being cast to ?. Consider the following $\mathbf{Ob}^{?}_{<:}$ expression:

$$\langle [l : \cdots] \Leftarrow ? \rangle \langle ? \Leftarrow [] \rangle [l = \cdots]$$

The type and implementation of the l method is unimportant, and so elided. The term is considered a *bad cast*, as the casts cannot be merged with a reduction step despite the fact that the term is well-typed. The merge reduction rule requires that the target type be a consistent super-type of the source, but in the given term the target is a sub-type, not a super-type.

Despite the cast's failure, the assumptions of the target type are satisfied by the object that is the body of the cast. The only reason for the cast's failure is that only the consistency of the types in the casts are considered, never the body of the cast itself. Because the source type of the cast is a strict super-type of the underlying object, information has been lost in the cast, and it is this loss of information that ultimately causes the cast to fail when it would otherwise be safe to proceed with a merge operation.

We can construct a program in the gradual source language that, following the cast insertion procedure, reduces to the given $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ term. The information loss is encoded using subsumption to forget the exact type of the object before casting it back to its original type. First, consider a metafunction id that takes a term and a type, and applies the term to an identity method:

$$\mathrm{id}(t, T) = [m = T\ \varsigma(x : T)\ x].m(t)$$

Any term generated by id is well-typed so long as the given type is a consistent super-type of the type of the term. The metafunction can be used to forget or assume information about the type of the term, either through subsumption or the application of a cast (generated by the cast insertion procedure).

An $\mathbf{Ob}^{?}_{<:}$ program that, following cast insertion, reduces to the bad cast above is the application of a method that accepts a value of unknown type and casts it to the ultimate target type of the cast, applied to the cast's body whose type is forgotten by subsumption:

$$[m = [l : \cdots]\ \varsigma(x : ?)\ \mathrm{id}(x, [l : \cdots])].m(\mathrm{id}([l = \cdots], []))$$

The application of id inside of the object is used to add an assumption that the type of the otherwise unknown parameter x has the type $[l = \cdots]$, required by the return type of the surrounding method: this assumption will

cause a cast to be inserted. The other application of id uses a simple super-type of the actual type of the object, so the type information is lost purely by subsumption and no cast is generated.

The $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ term generated by applying cast insertion to the $\mathbf{Ob}^{?}_{<:}$ term above is:

$$[m = [l : \cdots]\ \varsigma(x : ?)\ \mathrm{id}(\langle [l = \cdots] \Leftarrow ? \rangle\, x, [l : \cdots])].m(\langle ? \Leftarrow [] \rangle\, \mathrm{id}([l = \cdots], []))$$

The cast inside of the application of id is the result of the method application generated by id, whereas the cast in the argument of the call to m appears because the type of the parameter of m is $?$. The reason for the bad cast is that the source type of the latter cast is $[]$, since the more specific type of the body has been forgotten through subsumption.

The cast insertion procedure does not insert casts that only encode information loss from subsumption. The type system does not permit such a cast: the source and target types of a cast must be consistent, not consistent sub-types, so no subsumption can be present in the difference of the two types in a cast. Without interrogating the body, encoding subsumption into casts is necessary to prevent otherwise correct casts failing because of a loss of information, which would mean that any use of subsumption would cause a cast to appear in the run-time program.

A coercion in Mold that correctly describes the shallow structure of its subject cannot fail, because the match form actually interrogates the underlying object. Encoding subsumption in casts is not necessary, as all of the relevant information is available directly in the value of the object itself, with the trade-off that parameter types must be retained in every method definition. We believe this shows that object inspection should be a part of all cast calculi.

4.8 Boyland's example now meets the guarantee

Written in Mold, Boyland's typecase example does not violate the gradual guarantee, since the language's match construct does not examine the annotations on methods. The trade-off is that an Image and a Cowboy cannot be immediately distinguished between, so the second case is irrelevant.

$$\mathsf{o} \ni \mathsf{draw}\,\{\,z \to 1\,\}\,\{\,z \to z \ni \mathsf{draw}\,\{\,z \to 2\,\}\,\{\,z \to 3\,\}\,\}$$

The value of this expression cannot be 2, no matter what the value of o is, because that branch requires that o both contains and does not contain a method named draw.

Using casts, we can make an assumption that an object is either an Image or a Cowboy, such as in:

$$\mathsf{o} : \{\,\mathsf{draw} : \mathsf{Canvas} \to \top\,\}$$

The assumption can only be invalidated by requesting draw, passing a canvas object that does not satisfy the true parameter type of Nat. Changing the parameter type on the draw method of o can only affect the program semantics by raising an exception, which falls within the bounds of the gradual guarantee.

Mold does not include a dynamic type, but we postulate a precision relation $\sqsubseteq$ between stores and terms that is similar to the same relation in gradual languages. $t_1 \sqsubseteq t_2$ holds if the two terms are equal except that all of the type annotations in t_1 are super-types of the corresponding annotations in t_2, and the same for stores $\sigma_1 \sqsubseteq \sigma_2$.

Theorem 4.2 (Precision does not affect matching). *For any Mold stores σ_1 and σ_2 where $\vdash \sigma_1 : \Gamma_1$ and $\vdash \sigma_2 : \Gamma_2$, and values v_1 and v_2 where $\Gamma_1 \vdash v_1 : T_1$ and $\Gamma_2 \vdash v_2 : T_2$, if $\sigma_1 \sqsubseteq \sigma_2$ and reducing σ_2 and v_2 in a match $\sigma_2 \mid v_2 \ni m\,\{\,z \to t_1\,\}\,\{\,z \to t_2\,\} \longmapsto \sigma_2 \mid t_3$, then $\sigma_1 \mid v_1 \ni m\,\{\,z \to t_1\,\}\,\{\,z \to t_2\,\} \longmapsto \sigma_2 \mid t_3$.*

This property is obvious from the definitions of Rules E-Fst and E-Snd, which do not inspect the type annotations on the subject of the match.

We can then attempt to generalise this property to all of evaluation, effectively encoding the gradual guarantee.

Proposition 4.3 (Precision does not affect evaluation). *For any Mold stores σ_1 and σ_2 where $\vdash \sigma_1 : \Gamma_1$ and $\vdash \sigma_2 : \Gamma_2$, and terms t_1 and t_2 where $\Gamma_1 \vdash t_1 : T_1$ and $\Gamma_2 \vdash t_2 : T_2$, if $\sigma_1 \sqsubseteq \sigma_2$, $t_1 \sqsubseteq t_2$, and $\sigma_2 \mid t_2 \longmapsto \sigma_2' \mid t_2'$ where t_2' is not a raise, then $\sigma_1 \mid t_1 \longmapsto^* \sigma_1' \mid t_1'$ where $t_1' \sqsubseteq t_2'$.*

In the formal gradually-typed languages the guarantee has been applied to, using the structural typecase presented in Mold would be sufficient to uphold the guarantee. In contrast, Mold attempts to encode a more realistic model of a practical language, and as a result *does not* uphold the proposition above, as it is subject to the same loopholes of practical languages that we discuss in the next section.

5 THE GRADUAL GUARANTEE IS BROKEN

So far we have shown that our proposed shallow semantics allows the original example from Boyland (2014) to match the guarantee. Despite the suggestions of the original paper, neither structural typing, nor typecase, preclude the guarantee from holding. We will now show that, regardless, the guarantee as written excludes virtually all practical (supposedly) gradually-typed languages.

Formal models of gradual typing conventionally represent well-typed programs as either reducing successfully to a value, or resulting in a dynamic type error where untyped code can be blamed. As no plan survives first contact with the enemy, real-world implementations of gradual typing are more nuanced. Because immediate fatal errors are frowned upon by users, virtually all language implementations where dynamic type errors may occur include some facility for trapping and recovering from these errors. This facility is sufficient to recover full typecase behaviour, even from a language that otherwise lacks it.

An example will illustrate our claim. Consider this simple program, which distinguishes whether an argument of unknown type is a String or a Number:

```
String classify(o) {
  try {
    assertString(o);
    return "string";
  } catch (TypeError t) {
    try { assertNumber(o); return "number"; }
    catch (TypeError t2) { return "other"; }
  }
}
void assertString(String x) {}; void assertNumber(Number x) {};
```

The two method definitions at the bottom require that their parameter be a String or a Number, respectively. Calling these methods with the correct type will succeed, doing nothing; calling them with an invalid argument will raise a catchable type error.

The classify method first asserts that the unknown object is a String. If there is no error, it must have been a string, so it returns the description "string". Otherwise, it tries again, asserting that the object is a Number, with similar results. The method thus computes a typecase on its argument, without using any type-case construct, merely by using the common facility for detecting dynamic type errors. If the type annotation is removed from assertString, things change. There will no longer be a type error raised, regardless of the argument given. The method will universally return "string", even if given a number. This clearly violates the gradual guarantee: classify(1) has a different result depending on the presence or absence of a type annotation.

While this is an artificial example, the trap could in reality be far from the inciting code and have no direct intention to introspect on the object: nonetheless, the program behaves differently according to the type. The dynamic errors and the ability to detect them allow programs to behave differently according to type annotations.

These are standard features in real-world gradually-typed languages. Both Typed Racket (Takikawa et al. 2012) and Reticulated Python (Vitousek et al. 2014) incorporate the necessary features, and Vitousek et al. (2014) even go to some effort to ensure that the *right* catchable errors are raised.

We will demonstrate these violations of the guarantee by a cross-section of real languages from different traditions in the following sections, and then in Section 6 propose an updated Gradual Guarantee that current languages can satisfy, and alternative fixes to the criteria.

5.1 Reticulated Python

Reticulated Python (Vitousek et al. 2014), as a superset of Python, includes the standard try/except/finally exception handling of the language. Run-time type errors as a direct result of parameter annotations raise Exceptions. We can thus distinguish the presence of annotations straightforwardly, and produce different results accordingly. Consider the following Reticulated Python program:

```
def classify(o):
    try:
        assertString(o)
        return “string”
    except:
        return “not string”
def assertString(x : str):
    pass
print(classify(1))
```

The classify and assertString functions are analogous to the same-named methods from Section 5. At present, the code above will output “not string” because 1 is not an instance of the str type and an error will be raised, and then caught. If the str annotation is removed from the parameter in assertString, however, no error is raised. The first return statement will execute, and the program will output “string”. This happens under all three runtime semantics (transient, guarded, and monotonic).

Reticulated Python also supports more complex type annotations. The following program expects a list of floats to be provided, and gives different results depending on the presence of annotations.

```
def u(l : List(float)):
    for x in u:
        pass
def isFloatList(l):
    try:
        u([1, “string!”])
        return 1
    except CastError:
        return −1
    except:
        return −2
print(isFloatList([1, “string!”]))
```

With no annotation on u, the isFloatList method always returns 1. With the annotation, the program behaves differently under all three runtime semantics:

- Under the **monotonic** semantics, it returns −1. A specific CastError is raised as soon as u is entered, with all values of the list being checked on entry to the function.
- Under the default **transient** semantics, it returns −2. A generic exception is thrown as soon as x is bound to “string!” on the second iteration of the list.

- Under the **guarded** semantics, it returns 1 as in the unannotated case. No check is ever performed, because the loop variable x is never used.

Reticulated Python also includes class and structural object types, an explicit Dyn unknown type, and other features that allow more abstruse kinds of dynamic errors to arise when annotations are removed from parts of the program. The different semantics also allow errors to be triggered and caught potentially far from their original cause. We omit examples of these for the present, having shown that removing annotations can cause results of function calls to change.

Python, as a strongly-typed language, already had a range of run-time type errors that could be raised upon using an incorrectly-typed value (for example, adding a number and string). Reticulated Python ensures that its guarded semantics preserves subtyping relationships as far as possible to ensure that these errors, or isinstance checks, function as they would in untyped Python.

5.2 ECMAScript derivatives

Several languages deriving from, or building closely on top of, ECMAScript/JavaScript have gradual features. Dart, TypeScript, and ActionScript are the most prominent of these.

5.2.1 Dart. Dart's philosophy is that types should not affect program behaviour, in line with the Gradual Guarantee. Dart offers a deliberately unsound gradual type system by default, but can be executed in a slower "checked mode" where types are checked at run time (Bracha 2015). Type errors in checked mode result in standard Dart (and JavaScript) exceptions, which can be caught and distinguished as TypeErrors.

Exactly as for Reticulated Python, a straightforward try-catch recovers the presence or absence of the type annotation:

```
assertString(String x) {}
classify(x) {
  try {
    assertString(x);
    return "string";
  } catch(e) {
    return "not string";
  }
}
void main() { print(classify(1)); }
```

An interesting case with Dart is that this program behaves differently not only with or without type annotations, but inside and outside of the checked mode. When executed in unchecked mode (which is the default), no error is raised and the classify function returns "string" every time.

In an informal analysis of real-world open-source Dart code we found no clear instances of code intending to do this, outside of Dart's own test suite. Because Dart code usually runs in unchecked mode, and Dart philosophically follows the policy that types should not affect dynamic behaviour, it would be unusual and unreliable to do so. There are numerous instances, however, of code that in practice can behave differently in checked and unchecked mode because of undifferentiated catch clauses that trap all exceptions.

5.2.2 TypeScript. TypeScript, like Dart, has no run-time checks and is unsound, but unlike Dart lacks even a checked mode. Type annotations do not affect run-time behaviour of TypeScript code at all, but incorrectly-typed code may continue running until it becomes impossible to do so.

5.2.3 ActionScript. ActionScript is a superset of ECMAScript (JavaScript) with nominal class types, optional gradual annotations, and both static and dynamic type checking (Adobe Systems Incorporated 2008b). Run-time type errors are catchable instances of the ECMAScript TypeError (Adobe Systems Incorporated 2008a). Exactly analogously with Dart, handling this error can differentiate type annotations. ActionScript also includes an is primitive that performs a dynamic type test on the nominal types of an object.

5.3 Typed Racket

Typed Racket enables gradual typing on a module-by-module basis (Tobin-Hochstadt et al. 2011). Within a typed module, omitted type annotations are inferred from their initialization, or given the top type Any.

Consider the following typed module:

```
#lang typed/racket
(provide assertString)
(define (assertString [x : String]) : String x)
```

And this untyped module that imports it:

```
#lang racket
(require "typed.rkt")
(define (classify x)
  (with-handlers ([exn:fail:contract? (λ (e) "not string")])
    (assertString x)
    "string"))
(displayln (classify 1))
(displayln (classify "x"))
```

The classify function will correctly determine whether its argument is a String or not. If the type annotation on assertString is removed, the inferred parameter type of Any will not cause any error to be raised. If the typed module is also turned untyped with #lang racket, corresponding to the total removal of type annotations, the function will always return "string".

Typed Racket uses the guarded semantic, with special interposition support to preserve object identity (Strickland et al. 2012). A value passing through typed code has contracts attached to the reference, and equivalent contracts are applied to typed functions exported to untyped code. Within typed code, the static checker has resolved potential type errors and dynamic checking is not performed. As in Dart, the run-time checks are noticeably expensive compared to purely untyped code, but Racket by contrast prefers safety over efficiency. For function-typed parameters, Typed Racket has the shallow consistency behaviour we gave in Section 3.4. An error is raised immediately if the arity is wrong, but otherwise the function itself raises an error when given an incorrect argument.

5.4 Hack and HHVM

Hack is a gradually-typed variant of PHP that executes on HHVM (Adams et al. 2014). Hack is principally typechecked statically, but limited run-time checks do occur. HHVM will throw a catchable fatal error if a parameter or return type annotation is violated at a shallow level (HHVM Project 2016). A *catchable fatal error* is not an exception, and instead requires enabling a global error-handling hook in order not to terminate execution. This hook can throw an exception, or perform whatever other processing is desired; once caught, the error is not necessarily fatal. This example illustrates Hack/HHVM's behaviour:

```
function errorhandler($errno, $errstr, $errfile, $errline) {
```

```
    if ($errno == E_RECOVERABLE_ERROR) {
      print "not ";
      return true;
    }
    return false;
  }
  set_error_handler('errorhandler');
  function assertString(string $x) {}
  assertString(1);
  print "string";
```

In this case, it outputs "not ", and returns true to indicate that execution should continue.

The PHP language from which Hack derives has historically been happier with fatal runtime errors than most languages. Many categories of error have the effect of immediately terminating the script with an optional diagnostic message, while others provide the diagnostic and allow execution to continue. With this background, HHVM's use of a fatal error is consistent with existing practice, but the uniquely-PHP concept of a *catchable* fatal error makes type annotations recoverable anyway.

5.5 C♯ and Visual Basic

While the .Net platform is statically-typed, the Dynamic Language Runtime (Hugunin 2007) and platform extensions allow dynamic typing to work. Both the flagship languages of .Net include some gradual features.

5.5.1 Visual Basic. While not conventionally considered a gradually-typed language, Visual Basic.Net in its default non-strict mode allows any method call or conversion on the Object type, making it essentially a dynamic unknown type. Passing an Object to a subroutine or function expecting another type is statically permitted, and the type checked at run time. A Try/Catch construct allows trapping these errors.

Historically, Visual Basic included a Variant type with similar behaviour (Stuple and Stroo 1997), but the language had significantly more limited error recovery. It was possible, by use of an On Error Goto ... statement to detect and respond when a type error had occurred. Visual Basic for Applications, which continues to be embedded in Microsoft Office, retains this level of functionality.

5.5.2 C♯. C♯ is principally a statically-typed language, but now includes an explicit dynamic type that is statically compatible with other types and resolves usages dynamically. C♯ includes static type-based overloading in any case, so the Gradual Guarantee is intentionally not met. C♯ allows dynamic type errors to be trapped similarly to the other C-like languages discussed.

6 SAVING GRADUAL TYPING

We have seen that the Gradual Guarantee does not reflect real-world gradual languages, and that practical languages do not meet the current refined criteria for gradual typing of Siek et al. (2015). As a result there are *no* gradual languages at present. There are multiple ways this issue could be addressed. Our preference is for a new, weaker, guarantee that existing languages can satisfy, which we will lay out now.

6.1 A new Gradual Guarantee

We propose to update the guarantee from Boyland (2014) to address the issues we have reported:

> If an expression e_1 **containing no traps for failed typecasts**, evaluates without error in one step to e_2, then any e_1' where $e_1' \sqsubseteq e_1$ also evaluates in zero or more steps to e_2' where $e_2' \sqsubseteq e_2$.

The restriction on trapped typecasts is required in all languages, while in languages with higher-order typecases that do not follow our semantic from Section 3 an additional prohibition on typecase is also required.

Typed Racket, Reticulated Python, and other implementations of gradually-typed languages are able to satisfy this guarantee, while they did not meet the original. Our formal calculus from Section 4 is also compatible with this updated guarantee.

Nonetheless, this is a significant restriction of the original guarantee: at the language level it is no longer the case that type annotations cannot affect the behaviour of the program. It is not automatically "safe" to add or remove a type. In a formal model, with no free variables, or use of functions defined outside the expression, local analysis on this property is sufficient to know whether the annotation is in a safe context or not. In a practical system, a programmer using library code may not have any way of knowing what impact there might be.

6.2 An additional criterion for gradual typing

If the new guarantee from the previous section is unsuitable, the alternative is to have implementations of gradual typing adhere to the behaviour of the formal models: all dynamic type errors must be immediately, uncatchably, and unrecoverably fatal. With this additional criterion, the current gradual guarantee holds, but neither Typed Racket nor Reticulated Python are gradual. Our formal model can be made compatible with this criterion simply by removing the $t \Uparrow \{ z \to t \}$ term from the syntax: catching errors is a separable and removable feature.

The fact that current implementations do not do so already, despite the theory, appears to suggest that implementors, or their users, find such an arrangement undesirable. However, it is compatible with the broad principles of gradual typing from the software-engineering side: the programmer can add types over time, get diagnostics from partial executions that fail, and use partially-annotated programs indefinitely.

Such a criterion would eliminate a range of current dynamic languages from ever being made gradual, because they incorporate these features as a core component of the language. For example, Python is one such language. A common principle in Python is that it is "Easier to ask for forgiveness than permission" (EAFP): if there is something uncertain to be done, perform it speculatively and recover from the error if it fails. Python codebases are riddled with cases where, for example, an argument that may be of one of multiple types has a method called on it that exists on only some of those types, wrapped in a try-catch; or an argument is passed to the int function (which requires a string-like or numeric argument), even though it may be a complex object the value is to be extracted from. If all type errors are fatal, these standard idioms of the language become impossible.

6.3 No guarantee

As a final option, if both solutions from the previous sections are undesirable, it is the Gradual Guarantee itself that must go. In many circumstances it is clear that programmers, and even (gradual!) language designers, *want* to affect the dynamic semantics of the program with types. It is possible that the Guarantee is simply an undesirable property in itself, though we are not putting that argument forward.

6.4 Resolving the conflict

Either of our proposals from Section 6.1 and 6.2 suffice to recover from the untenable position of the current refined criteria for gradual typing of Siek et al. (2015). We regard requiring unrecoverable errors as undesirable from a reliability and expressiveness perspective, and prefer to adopt a slightly weaker gradual guarantee, but there are reasons to follow either path.

In either case, the real-world languages do not presently meet the criteria for gradual typing. With a new guarantee, they would immediately do so, while with an additional criterion they would need modification to remove the offending functionality.

7 RELATED WORK

Bloom et al. (2009)'s "like types" are still laxer than our structural type testing system, with any value satisfying a like type at run time. Only actually calling a missing method will fail. As in gradual systems, concrete return and parameter types are checked.

Gradual typing for object-oriented languages was first considered by Siek and Taha (2007), but most of the literature on formal gradual type systems is functional (Cimini and Siek 2016; Garcia 2013; Wadler and Findler 2009). Garcia et al. (2016) included records and structural types in their theoretical rebasing of the discipline, introducing a *gradual row* to mix known and unknown structural type information for width typing. Takikawa et al. (2012) implemented gradual typing for first-class classes, though Typed Racket uses a 'macro' approach to gradual typing that does not include the dynamic type (Tobin-Hochstadt and Felleisen 2008), in contrast to the 'micro' approach of Siek and Taha (2006).

Object-oriented approaches to gradual typing in Python have been explored by Siek et al. (2012) and Vitousek et al. (2014), resulting in the Reticulated Python language, which we have already discussed.

The syntax of casts in Mold are most like the coercions of Henglein (1994), modernised by Herman et al. (2010) and explored by Garcia (2013). The application of union types to gradual typing also appears in the work of Siek and Tobin-Hochstadt (2016), using the traditional gradual typing cast syntax. Unlike both coercions and traditional gradual typing casts, casts in Mold include neither the source type or target type of the cast, only presenting a refinement of the existing type of the cast's body. Both casts and coercions in the literature of gradual typing evaluate their assumptions by merging casts together, and failing if the resulting cast makes invalid assumptions; in contrast, merging Mold casts can never fail, and only a match can discover that an assumption has failed.

7.1 Incomplete formal models

The formal calculi of gradual typing were consistent with the guarantee, despite the languages they attempted to model being actively inconsistent. Overly-simplified formal models that omit relevant features, and the *interactions* of those features, for the properties they seek to prove are unhelpful. Recently a very similar problem has been illustrated dramatically by Amin and Tate (2016)'s demonstration that Java's type system was unsound, despite many years of formal modelling consistently proving soundness of subsets, because the necessary features (null pointers and generic types) were never included in the same model. Similarly, Summers (2009) noted that usefully modelling Java required state. We encourage more realistic models so that such issues can be discovered sooner. Our calculus, Mold, includes both object inspection and error recovery specifically for this purpose.

8 CONCLUSION

The original Gradual Guarantee paper (Boyland 2014) suggested that structural type tests were incompatible with a gradually-typed language. We have shown a method for implementing structural type tests without the possibility of type annotations affecting the result of the program. Our method satisfies the Gradual Guarantee and uses the existing features of a gradually-typed language to retain type safety. We have presented a formal model of this system, and noted a number of real-world implementations that have fallen into it inadvertently and should not try to get out.

We have also shown that the original Gradual Guarantee is inconsistent with the design of real-world gradual languages. In practical languages, such as Typed Racket and Reticulated Python, run-time type errors are catchable and execution of the program can continue, which is all that is required for type annotations to have impact on the result of the program. These languages are thus, according to the Guarantee, not gradual after all. Our proposed replacement Gradual Guarantee accepts the facts on the ground and limits the guarantee to those (part-) programs where these features are unused.

REFERENCES

Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The Hiphop Virtual Machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 777–790. DOI:http://dx.doi.org/10.1145/2660193.2660199

Adobe Systems Incorporated. 2008a. Adobe ActionScript 3.0 * Type Checking. http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7f8a.html. (2008).

Adobe Systems Incorporated. 2008b. *Programming Adobe ActionScript 3.0*.

Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 838–848. DOI:http://dx.doi.org/10.1145/2983990.2984004

Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. 2009. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 117–136. DOI:http://dx.doi.org/10.1145/1640089.1640098

John Tang Boyland. 2014. The Problem of Structural Type Tests in a Gradual-Typed Language. In *FOOL*. ACM, New York, NY, USA.

Gilad Bracha. 2015. *The Dart Programming Language* (1st ed.). Addison-Wesley Professional.

Kim Bruce, Andrew Black, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. 2013. Seeking Grace: a new object-oriented language for novices. In *Proceedings 44th SIGCSE Technical Symposium on Computer Science Education*. ACM, 129–134. DOI:http://dx.doi.org/10.1145/2445196.2445240

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*. 443–455. DOI:http://dx.doi.org/10.1145/2837614.2837632

Ronald Garcia. 2013. Calculating Threesomes, with Blame. In *Proceedings of the 18th International Conference on Functional Programming (ICFP'13)*. 417–428. DOI:http://dx.doi.org/10.1145/2500365.2500603

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*. 429–442. DOI:http://dx.doi.org/10.1145/2837614.2837670

Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (1994), 197–230. DOI:http://dx.doi.org/10.1016/0167-6423(94)00004-2

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 2 (2010), 167–189. DOI:http://dx.doi.org/10.1007/s10990-011-9066-z

HHVM Project. 2016. Hack documentation: Types: Runtime. https://docs.hhvm.com/hack/types/runtime. (2016).

Michael Homer. 2016. Kernan. http://gracelang.org/applications/grace-versions/kernan/. (2016).

Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns as objects in Grace. In *Proceedings of the 8th symposium on Dynamic languages (DLS '12)*. ACM, New York, NY, USA, 17–28. DOI:http://dx.doi.org/10.1145/2384577.2384581

Jim Hugunin. 2007. Bringing Dynamic Languages to .NET with the DLR. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. ACM, New York, NY, USA, 101–101. DOI:http://dx.doi.org/10.1145/1297081.1297083

Timothy Jones. 2015. Hopper. https://www.npmjs.com/package/hopper. (2015).

Timothy Jones and James Noble. 2014. Tinygrace: A Simple, Safe and Structurally Typed Language. In *FTFJP*. ACM, New York, NY, USA.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 31st International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. 624–641. DOI:http://dx.doi.org/10.1145/2983990.2984008

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

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*. Springer-Verlag, 2–27. http://dl.acm.org/citation.cfm?id=2394758.2394762

Jeremy G. Siek and Sam Tobin-Hochstadt. 2016. *The Recursive Union of Some Gradual Types*. DOI:http://dx.doi.org/10.1007/978-3-319-30936-1_21

Jeremy G. Siek, Michael M. Vitousek, and Shashank Bharadwaj. 2012. Gradual Typing for Mutable Objects. Unpublished manuscript. (2012). https://ecee.colorado.edu/~siek/gtmo.pdf

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL*. 274–293. DOI:http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.274

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*. 943–962. DOI:http://dx.doi.org/10.1145/2384616.2384685

Stuart J. Stuple and Eric Stroo (Eds.). 1997. *Microsoft Visual Basic 5.0 Programmer's Guide*. Microsoft Press.

Alexander J Summers. 2009. Modelling Java requires state. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*. ACM, 10.

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-class Classes. In *OOPSLA*.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Symposium on Principles of Programming Languages (POPL'08)*. 395–406. DOI:http://dx.doi.org/10.1145/1328438.1328486

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *PLDI*. 10.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *DLS*. 45–56. DOI:http://dx.doi.org/10.1145/2661088.2661101

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *ESOP*. 1–16. DOI:http://dx.doi.org/10.1007/978-3-642-00590-9_1

A EXTENDED JUDGMENTS

$\boxed{\vdash T}$ (W-Uni)
$$\frac{\overline{\vdash S}}{\vdash \overline{S}}$$

$\boxed{\vdash S}$ (w-Str)
$$\frac{\overline{\text{identify}(D)\ \text{unique}} \qquad \overline{\vdash D}}{\vdash \{\, \overline{D} \,\}}$$

$\boxed{\vdash D}$ (W-Sig)
$$\frac{\vdash T_1, T_2}{\vdash m : T_1 \to T_2}$$

Fig. 5. Type well-formedness judgment

$\boxed{T <: T}$ (S-Uni)
$$\frac{\forall S_i.\ \exists S_j.\ S_i <: S_j}{\overline{S_i} <: \overline{S_j}}$$

$\boxed{S <: S}$ (S-Str)
$$\frac{\forall D_j.\ \exists D_i.\ D_i <: D_j}{\{\, \overline{D_i} \,\} <: \{\, \overline{D_j} \,\}}$$

$\boxed{D <: D}$ (S-Sig)
$$\frac{T_3 <: T_1 \qquad T_2 <: T_4}{m : T_1 \to T_2 <: m : T_3 \to T_4}$$

Fig. 6. Subtyping judgment

$\cup : \textsc{Type} \times \textsc{Type} \to \textsc{Type}$
$\overline{S_i} \cup \overline{S_j} = (\overline{S_i}, \overline{S_j})$

$\cap : \textsc{Type} \times \textsc{Type} \to \textsc{Type}$
$\overline{S_i}^{\,i \le n} \cap \overline{S_j}^{\,j \le m} = \overline{S_i \cap S_j}^{\,(i,j) \le (n,m)}$

$\cap : \textsc{Struct} \times \textsc{Struct} \to \textsc{Struct}$
$\{\, \overline{D_i} \,\} \cap \{\, \overline{D_j} \,\} = \{\, (\overline{D_i} \cap \overline{D_j}) \,\}$

$\cap : \textsc{Seq}(\textsc{Decl}) \times \textsc{Seq}(\textsc{Decl}) \to \textsc{Seq}(\textsc{Decl})$
$(D_1, \overline{D_i}) \cap (\overline{D_j}, D_2, \overline{D_k}) = (D_1 \cap D_2), (\overline{D_i} \cap (\overline{D_j}, \overline{D_k}))$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ **where** $\text{identify}(D_1) = \text{identify}(D_2)$
$(D_1, \overline{D_i}) \cap \overline{D_j} = D_1, (\overline{D_i} \cap \overline{D_j})$
$\cdot \cap \overline{D_j} = \overline{D_j}$

$\cap : \textsc{Decl} \times \textsc{Decl} \rightharpoonup \textsc{Decl}$
$m : T_1 \to T_2 \cap m : T_3 \to T_4 = m : (T_1 \cup T_3) \to (T_2 \cap T_4)$

Fig. 7. Type combinators

$- : \textsc{Type} \times \textsc{Var} \to \textsc{Type}$
$\overline{S_i} - m = \overline{S_i - m}$

$- : \textsc{Struct} \times \textsc{Var} \rightharpoonup \textsc{Type}$
$$\{\, \overline{D} \,\} - m = \begin{cases} m \in \overline{\text{identify}(D)} & \text{undefined} \\ m \notin \overline{\text{identify}(D)} & \{\, \overline{D} \,\} \end{cases}$$

Fig. 8. Signature subtraction

B PROPERTIES

Lemma B.1 (Union identity with $\bot$).

$$\overline{T \cup \bot = T}$$

Proof. Immediate from the definition of $\cup$, since $\bot$ contains no structural types in its union. □

Lemma B.2 (Intersection identity with $\top$).

$$\overline{T \cap \top = T}$$

Proof. Results in distributing $S \cap \{\}$ where S are the structural types in the union of T. $S \cap \{\} = S$ is immediate from the definition of $\cap$, since $\{\}$ contains no signatures. □

Lemma B.3 (Subtraction identity if m is absent).

$$\frac{T = \{\overline{D}\} \qquad \overline{m \notin \overline{identify(D)}}}{T - m = T}$$

Proof. Immediate from the definition of subtraction. □

Lemma B.4 (Subtyping is reflexive).

$$\frac{\vdash T}{T <: T}$$

Proof. Trivial induction over the proof that $\vdash T$. □

Lemma B.5 (Subtyping is transitive).

$$\frac{\vdash T_1, T_2, T_3 \qquad T_1 <: T_2 \qquad T_2 <: T_3}{T_1 <: T_3}$$

Proof. Simple mutual induction over the proofs that $T_1 <: T_2$ and $T_2 <: T_3$. □

Lemma B.6 ($\bot$ is the bottom of the subtyping lattice).

$$\overline{\bot <: T}$$

Proof. $\overline{S_i}$ is empty, so Rule S-Uni is immediate. □

Lemma B.7 ($\{\}$ is the top of the structural subtyping lattice).

$$\overline{S <: \{\}}$$

Proof. $\overline{S_j}$ is empty, so Rule S-Str is immediate. □

Lemma B.8 ($\top$ is the top of the subtyping lattice).

$$\overline{T <: \top}$$

Proof. For every S in the union of T, $S <: \{\}$ by Lemma B.7, so Rule S-Uni applies. □

Lemma B.9 ($\bot$ is the unique bottom type).

$$\frac{T <: \bot}{T = \bot}$$

Proof. By Rule S-Uni, for every structural type in T, there must be a corresponding structural type in $\bot$. Since $\bot$ contains no structural types, the rule can only hold if T also contains no structural types, and so is also $\bot$. □

Lemma B.10 (Structural subtype implies declaration subtype).

$$\frac{\{\,\overline{D_i}\,\} <: \{\,D\,\}}{\exists D_i.\ D_i <: D}$$

Proof. Simple case analysis of the proof that $\{\,\overline{D_i}\,\} <: \{\,D\,\}$. □

Lemma B.11 (Declaration subtype implies identifier equality).

$$\frac{D_1 <: D_2}{\mathit{identify}(D_1) = \mathit{identify}(D_2)}$$

Proof. Immediate from Rule S-Sig. □

Lemma B.12 (Union produces super-type).

$$\frac{T_1 \cup T_2 = T_3}{T_1 <: T_3 \qquad T_2 <: T_3}$$

Proof. Every structural type in the unions of T_1 and T_2 appears in the union of T_3, so Rule S-Uni can satisfy both goals through Lemma B.4 (modified to apply to structural types, which is part of the proof for that lemma). □

Lemma B.13 (Intersection produces sub-type).

$$\frac{T_1 \cap T_2 = T_3}{T_3 <: T_1 \qquad T_3 <: T_2}$$

Proof. Intersection produces a union with every possible pairing of structural types across T_1 and T_2, and Rule S-Uni requires that we demonstrate that every one of these pairings has a corresponding structural super-type in both T_1 and T_2 in order to satisfy both of the goals: for any structural intersection $S_1 \cap S_2$, we can show that both S_1 and S_2 are structural super-types. By Rule S-Str, for every signature in both S_1 and S_2, there is some corresponding signature in the intersection of their sequences of signatures. In the case that the identifier of a signature in one sequence does not appear in the other sequence, this requirement is satisfied by Lemma B.4 (modified to apply to signatures, which is part of the proof for that lemma). If two signatures in either sequence have the same identifier, then we need to show that the intersection of those signatures is a sub-type of each signature: if $D_1 \cap D_2$ is defined, then both D_1 and D_2 are signature super-types. This requirement is satisfied by Rule S-Sig, coinductively applying Lemma B.12 for the parameter types and this lemma for the return type. □

Lemma B.14 (Subtraction produces sub-type).

$$\frac{T_1 - m = T_2}{T_2 <: T_1}$$

Proof. Any structural types the appear in the union of T_2 are unmodified from T_1; the subtraction only removes structural types. Rule S-Uni applies with Lemma B.4 (again applied to structural types). □

Lemma B.15 (Emptiness of $\bot$). $\Gamma \vdash v : \bot$ *cannot hold.*

Proof. Assume that $\Gamma \vdash v : \bot$ does hold, case analysis on the proof.

(T-Sub) This rule must select a type T_1 such that $T_1 <: \bot$. By Lemma B.9, $T_1 = \bot$, so the rule also contains a proof that $\Gamma \vdash v : \bot$.

(T-Cst) The body must also have type $\bot$, so the rule also contains a proof that $\Gamma \vdash y : \bot$.

The remaining rules do not apply to $\bot$. Since Rule T-Sub is defined inductively and is the only applicable rule, the fact that it must repeat infinitely forms a contradiction with the assumption that such a proof could exist. □

Lemma B.16 (Store typing domain equality).

$$\frac{\vdash \sigma : \overline{(x : T)}}{\overline{x} = dom(\sigma)}$$

Proof. Immediate from the judgment $\vdash \sigma : \Gamma$ by Rule T-Sto. □

Lemma B.17 (Store typing is structural).

$$\frac{\vdash \sigma : \Gamma \qquad y : T \in \Gamma}{T = \{\, \overline{D} \,\}}$$

Proof. Immediate from the judgment $\vdash \sigma : \Gamma$ by Rule T-Sto. □

Lemma B.18 (Environment lookup corresponds to store).

$$\frac{\vdash \sigma : \Gamma \qquad y : \{\, \overline{D} \,\} \in \Gamma}{\overline{d} = \sigma(y) \qquad \overline{signature(d) = D}}$$

Proof. By Rule T-Sto, the signatures assigned to the store are computed by Rule T-Sig, which assigns the exact signature of the definition, so any signature stored in the type of a reference y corresponds exactly the definition found at y in σ. □

Lemma B.19 (Canonicity of forms).

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : \{\, D \,\}}{d \in \sigma(y) \qquad signature(d) <: D}$$

Proof. By analysis of the proof that $\Gamma \vdash y : \{\, D \,\}$, proceeding through an arbitrary number of applications of Rule T-Sub, terminating in Rule T-Var where $\Gamma \vdash y : \{\, \overline{D_i} \,\}$. By Lemma B.18, $\overline{D_i}$ correspond exactly to the signatures of the definitions at $\sigma(y)$. The intervening subtyping premises of Rule T-Sub combine under Lemma B.5 to form a proof that $\{\, \overline{D_i} \,\} <: \{\, D \,\}$, so it follows by Lemma B.10 that one of the signatures D_i is compatible with D. □

Lemma B.20 (Canonicity of casts).

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash v : \{\, D \,\}}{d \in \sigma(v) \qquad identify(d) = identify(D)}$$

Proof. Immediate from the induction hypothesis applied to the inversion of Rule T-Cst, with the subtyping conclusion of Lemma B.19 implying the equality of identifiers. □

Lemma B.21 (Typing implies progress). *For any program $\langle \sigma, t \rangle$, if $\vdash \sigma : \Gamma$ and $\Gamma \vdash t : T$ then either:*

- $\exists v.\ t = v$
- $\exists v.\ t = \Uparrow v$
- $\exists \sigma'\ t'.\ \sigma \mid t \longmapsto \sigma' \mid t'$

Proof. By induction on the derivation of the proof that $\Gamma \vdash t : T$, with a case analysis on the last step.

(T-Var) By Lemma B.16, $t = y$, so immediate.
(T-Rse) $t = \Uparrow t'$, so immediate. Induction applies Rule E-Rse when this appears in the hole of a context F.
(T-Sub) Induction on the premise that $\Gamma \vdash t : T'$ for some T'.
(T-Obj) Rule E-Obj, the requirement of uniqueness of method identifiers is immediate from the premise of the assumption.
(T-Req) Induction on the typing of the subterms. If the receiver is a reference and all subterms are values, then Rule E-Req, with the premise satisfied by Lemma B.19 and Lemma B.11. If the receiver is a cast, then Rule E-Req no longer applies. Signature selection from a type $S(m) : T_1 \to T_2$ always succeeds by definition, and Lemma B.20 guarantees that a corresponding method appears in the store, so Rule E-Cst can be applied.
(T-Rsc) Induction on the typing of the body. If the body is a value, then Rule E-Sfe. If the body contains a raise in the hole of a context F (the only location where this is not handled immediately by Rule E-Rse), then Rule E-Rsc.
(T-Mch) By induction on the typing of the body. If the body is a reference y, then y appears in the domain of Γ by Lemma B.16, so either Rule E-Fst or E-Snd apply.
(T-Cst) Immediate from the induction hypothesis and either Rule E-Mrg if the body is a cast or Rule E-Cng for any other term, or the whole term is a value.

This covers all cases. □

Lemma B.22 (Substitution preserves typing).

$$\frac{\Gamma \vdash v : T_1 \qquad \Gamma_1, z : T_1, \Gamma_2 \vdash t : T_2 \qquad z \notin \Gamma_2}{\Gamma_1, \Gamma_2 \vdash [v/z]t : T_2}$$

Proof. By induction on the derivation of the proof that $\Gamma_1, z : T_1, \Gamma_2 \vdash t : T_2$, with a case analysis on the last step.

(T-Var) If $t = v$ then $T_1 = T_2$, so immediate from the proof that $\Gamma \vdash v : T_1$. Otherwise the variable has not been substituted and the proof remains unchanged.

The remaining rules all follow immediately from induction and applications of weakening; any binding that shadows z terminates the substitution. □

Lemma B.23 (Adding ground preserves typing).

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : T \qquad \overline{d} = \sigma(y) \qquad m \in \overline{identify(d)}}{\Gamma \vdash y : T \cap \{\, m : \bot \to \top \,\}}$$

Proof. By induction on the derivation of the proof that $\Gamma \vdash y : T$, with a case analysis on the last step.

(T-Var) T is a structural type $\{\, \overline{D_i} \,\}$ by Lemma B.17. Since $a \in \overline{\text{identify}(d)}$, by Lemma B.18 there is a D_i such that $\text{identify}(D_i) = a$. Intersecting the ground type of a signature with a structural type that contains that signature is an identity on the structural type, by Lemmas B.2 and B.1.
(T-Sub) By Lemma B.13 the result of the intersection is a subtype of the original type, so one more application of Rule T-Sub.

The remaining cases do not apply to y. □

Lemma B.24 (Subtraction preserves typing).

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash y : T \qquad \overline{d} = \sigma(y) \qquad m \notin \overline{identify(d)}}{\Gamma \vdash y : T - m}$$

Proof. By induction on the derivation of the proof that $\Gamma \vdash y : T$, with a case analysis on the last step.

(T-Var) T is a structural type $\{ \overline{D_i} \}$ by Lemma B.17. Since $a \notin \overline{\text{identify}(d)}$, by Lemma B.18 there is no D_i such that $\text{identify}(D_i) = m$. By Lemma B.3, $T - m = T$.

(T-Sub) By Lemma B.14 the result of the subtraction is a subtype of the original type, so one more application of Rule T-Sub.

The remaining cases do not apply to y. □

Lemma B.25 (Coercion implies typing).

$$\frac{\Gamma \vdash t : T_1}{\Gamma \vdash coerce(t, T_2) : T_2}$$

Proof. Case analysis of the input T_2:

$\bot$ The outcome is $\Uparrow t$, which has the type $\bot$ by Rule T-Rse and the existing proof that t is well-typed.

$\top$ The outcome is t, which has the type $\top$ by Rule T-Sub and the existing proof that t is well-typed.

else The outcome is $t \ni m \{ z \to t_1 \} \{ z \to t_2 \}$, so we can type this with Rule T-Mch if t_1 and t_2 are well-typed. Use of z is well-typed by Rule T-Var in the extended typing environment of the two blocks. The ground of cast signatures introduced in t_1 onto z are guaranteed to appear in the type of z by the surrounding match, and so can be typed by Rule T-Cst, with the remaining forms typed by induction. The typing of t_2 follows directly from induction.

This covers all cases. □

Lemma B.26 (Selection return subtypes lookup return).

$$\frac{S <: \{ m : T_1 \to T_2 \} \qquad S(m) : T_3 \to T_4}{T_4 <: T_2}$$

Proof. Mutual analysis of the two inputs and rebuilding the corresponding subtyping rules, or immediate if no signature m appears in S thanks to Lemma B.1. □

Lemma B.27 (Reduction preserves typing).

$$\frac{\vdash \sigma : \Gamma \qquad \Gamma \vdash t : T \qquad \sigma \mid t \longmapsto \sigma' \mid t'}{\vdash \sigma' : \Gamma' \qquad \Gamma' \vdash t' : T}$$

Proof. Induction on the derivation of the proof that $\sigma \mid t \longmapsto \sigma' \mid t'$, with a case analysis on the last step.

(E-Obj) It follows from the inversion of Rule T-Obj (and Rule T-Sub) that the definitions $\overline{d}$ in the object constructor are typed $\Gamma, z : T \vdash d : D$. By Lemma B.22, the definitions $[y/z]\overline{d}$ in the store at the newly allocated reference y can be typed with the same signatures $\overline{D}$, so the reference can be typed $\Gamma \vdash y : T$ by Rule T-Var.

(E-Req) In the result $[v/z]t$, the term t has come from the body of a definition with signature D in the store at the reference y, so Lemma B.19 applied to the proof that y has the type $\{ D \}$ means that, through inversion of Rule T-Var on the typing of y and Rule T-Sto on the proof that $\vdash \sigma : \Gamma$ (plus the guarantee

that the identifiers of the definitions in an object in the store are unique), it must be the case that $\Gamma, z : T_3 \vdash t : T_4$ where $T_1 <: T_3$ and $T_4 <: T_2$. Inversion of Rule T-Req (and Rule T-Sub) proves that $\Gamma \vdash v : T_1$. Lemma B.22 moves from the typing environment to substitution to prove that $\Gamma \vdash [v/z]t : T_4$, and Rule T-Sub applies to the proof that $T_4 <: T_2$ to prove that the result is typed T_2.

(E-Rse) Rule T-Sub applied to Rule T-Rse. The necessary subtyping is immediately satisfied by Lemma B.6.

(E-Sfe) It follows from the inversion of Rule T-Rsc (and Rule T-Sub) that the resulting value v is typed $\Gamma \vdash v : T_1$, where the type of the rescue was $T_1 \cup T_2$. Rule T-Sub and $T_1 <: T_1 \cup T_2$ (by Lemma B.12) combine to prove that $\Gamma \vdash v : T_1 \cup T_2$.

(E-Rsc) It follows from the inversion of Rule T-Rsc (and Rule T-Sub) that the resulting term $[v/z]t$ is typed $\Gamma, z : \top \vdash t : T_2$, where the type of the rescue was $T_1 \cup T_2$. Lemma B.22 moves from the typing environment to substitution to prove that $\Gamma \vdash [v/z]t : T_2$ by applying Rule T-Sub to the proof that $\Gamma \vdash v : T_1$ and the trivial subtyping $T_1 <: \top$. Rule T-Sub and $T_2 <: T_1 \cup T_2$ (by Lemma B.12) combine to prove that $\Gamma \vdash [v/z]t : T_1 \cup T_2$.

(E-Fst) It follows from the inversion of Rule T-Mch (and Rule T-Sub) that the resulting term $[v/z]t$ is typed $\Gamma, z : \text{ground}(m) \vdash t : T_2$, where the type of the match was $T_2 \cup T_3$. Lemma B.22 moves from the typing environment to substitution to prove that $\Gamma \vdash [v/z]t : T_2$ by applying Lemma B.23 to the proof that $\Gamma \vdash v : T_1$. Rule T-Sub and $T_2 <: T_2 \cup T_3$ (by Lemma B.12) combine to prove that $\Gamma \vdash [v/z]t : T_2 \cup T_3$.

(E-Snd) It follows from the inversion of Rule T-Mch (and Rule T-Sub) that the resulting term $[v/z]t$ is typed $\Gamma, z : T_1 - m \vdash t : T_3$, where the type of the match was $T_2 \cup T_3$. Lemma B.22 moves from the typing environment to substitution to prove that $\Gamma \vdash [v/z]t : T_3$ by applying Lemma B.24 to the proof that $\Gamma \vdash v : T_1$. Rule T-Sub and $T_3 <: T_2 \cup T_3$ (by Lemma B.12) combine to prove that $\Gamma \vdash [v/z]t : T_2 \cup T_3$.

(E-Cst) The inversion of Rule T-Cst (and Rule T-Sub) on the original cast proves that the value v has a type that subtypes the ground of the cast type T, and the type of the cast itself is the intersection of the type of v with T. The application of Lemma B.25 to the types of the arguments alongside the typing of v, whose relation with ground ensures that the relevant method appears in the type, rebuilds Rule T-Req for the request inside the resulting cast, and then an application of Rule T-Cst types the cast as the intersection of the return type of the inner request and the return type selected from T by Lemma B.25. The type is the lowest type on the lattice that could have been selected by the typing rules by Lemmas B.13 and B.26, so applying Rule T-Sub to this typing with Lemma B.26 provides the necessary type.

(E-Mrg) Rule T-Cst, with the resulting obligation that $T <: \text{ground}(S_1 \cap S_2)$, which is satisfied by the fact that no new signatures can be added by a cast and that ground signatures are trivial to subtype.

This covers all cases. □

Theorem 4.1 (Type soundness). *For any Mold program $\langle \sigma, t \rangle$, if $\vdash \sigma : \Gamma$ and $\Gamma \vdash t : T$, then either:*

- $\exists v.\ t = v$, *so* $\Gamma \vdash v : T$
- $\exists v.\ t = \Uparrow v$, *so* $\Gamma \vdash \Uparrow v : T$
- $\exists \sigma'\ t'.\ \sigma \mid t \longmapsto \sigma \mid t'$, *with* $\vdash \sigma' : \Gamma'$ *and* $\Gamma' \vdash t' : T$

Proof. Immediate from Lemmas B.21 and B.27. □