Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[2], Matteo Cimini[3], and John Tang Boyland[4]

1-3 Indiana University – Bloomington, School of Informatics and Computing
150 S. Woodlawn Ave. Bloomington, IN 47405, USA
`jsiek@indiana.edu`

4 University of Wisconsin – Milwaukee, Department of EECS
PO Box 784, Milwaukee WI 53201, USA
`boyland@cs.uwm.edu`

Abstract

Siek and Taha [2006] coined the term *gradual typing* to describe a technical approach to integrating static and dynamic typing within a single language that 1) puts the programmer in control of which regions of code are statically or dynamically typed, and 2) enables the gradual migration of code between the two typing disciplines. Since 2006, the term *gradual typing* has become quite popular but its meaning has become diluted to encompass anything related to the integration of static and dynamic typing. This dilution is partly the fault of the original paper, which provided an incomplete formal characterization of what it means to be gradually typed. In this paper we draw a crisp line in the sand, articulating a new formal property, named the *gradual guarantee*, that relates the behavior of programs that differ only with respect to the precision of their type annotations. We argue that the gradual guarantee provides important guidance for designers of gradually-typed languages. We survey the gradual typing literature, critiquing designs in light of the gradual guarantee. We report on a mechanized proof that the gradual guarantee holds for the Gradually-Typed Lambda Calculus.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs – Type structure

Keywords and phrases gradual typing, type systems, semantics, dynamic languages

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

We review four criteria for gradually-typed languages that appear in the literature and point out the need for a new criteria that characterizes what it means for a language to support the gradual migration of code between the static and dynamic type disciplines. We then introduce our new criteria, the *gradual guarantee*.

1.1 Gradual as a Superset of Static and Dynamic

A gradually-typed language tries to encompass both static and dynamic type checking, with the programmer controlling which regions of code are checked when by adding or removing type annotations. Thus, a gradually-typed language tries to be a superset of two languages: an untyped language and a typed language. For example, the gradually-typed lambda calculus (GTLC) [9] tries to encompass both the dynamically-typed lambda calculus (DTLC) and the simply-typed lambda calculus (STLC). Siek and Taha [9] prove that the GTLC type

* This work was partially supported by NSF grant 1360694.

system is equivalent to the STLC on fully-annotated programs. We extend this criteria to require the dynamic semantics of the GTLC to coincide with the STLC on fully-annotated programs in the theorem below. Let e range over the expressions of the GTLC and T range over its types: base types B, function types $T \to T$, and the unknown type $\star$ (aka. type dynamic). We say that a type is *static* if the unknown type does not occur in it.

▶ **Theorem 1** (Equivalence to the STLC for fully-annotated terms).
Suppose e is fully-annotated and T is static.
1. *e is well-typed at type T in the GTLC if and only if e is well-typed at T in the STLC. (Siek and Taha [9]).*
2. *e evaluates to the same result in the GTLC and STLC.*

The relation to the DTLC is more nuanced by necessity. Suppose the constant `inc` has type `Int`$\to$`Int` and the constant `true` has type `Bool`. Then the application of `inc` to `true` is ill-typed in the GTLC even though it is a valid program in the DTLC (with constants). Similar issues arise when extending the GTLC with other constructs, such as conditionals, where one must choose to favor either static typing or dynamic typing. Nevertheless, there is a simple encoding of the DTLC into the GTLC, written $[\![\cdot]\!]$, that casts all constants to the unknown type.

▶ **Theorem 2** (Embedding of DTLC).
Suppose that e is a term of DTLC.
1. *$[\![e]\!]$ is well-typed in the GTLC (Siek and Taha [9]).*
2. *e evaluates to the same result as $[\![e]\!]$.*

These two theorems only characterize programs at the two extremes: fully-annotated and completely un-annotated. They say nothing about partially-annotated programs, which is the norm for gradually-typed languages. However, before turning to address this issue we discuss one more criteria from the literature.

1.2 Soundness for Gradually-Typed Languages

One would also like gradually-typed language to be sound in some sense. Siek and Taha [9] prove that the GTLC is sound in the same way that many dynamically-typed languages are sound: execution never encounters untrapped errors (but trapped errors are ubiquitous).

▶ **Theorem 3** (Type Safety, Siek and Taha [9]). *If e is well-typed at type T, then either e evaluates to a value v and v has type T, or e evaluates to a trapped error, or e diverges.*

This theorem is unsatisfying because it does not tell us that statically-typed regions of code are free of trapped errors. Blame tracking [4] provides the right mechanism for formulating type soundness for gradually-typed programs. Blame tracking maps a trapped error back to the source location of the cast that caused the error (which is non-trivial for higher-order languages). The Blame Theorem [13, 17] characterizes the safe versus possibly unsafe casts. Adapting these results to gradual typing, we arrive at the following theorem that follows easily from the Blame Theorem.

▶ **Theorem 4** (Subtyping Theorem). *Given a program e of the GTLC, if the implicit casts (if any) at location ℓ respect subtyping, then e does not evaluate to a trapped error blaming ℓ.*

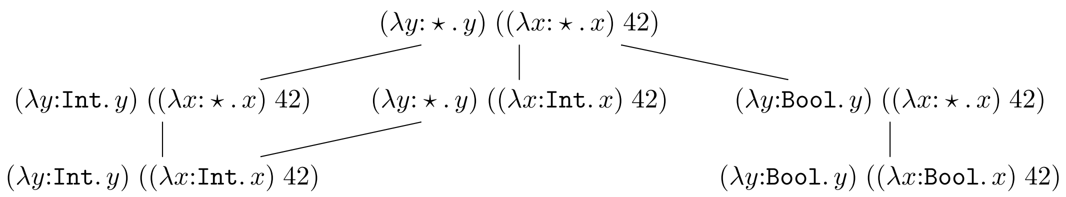
$(\lambda y{:}\star\,.\,y)\;((\lambda x{:}\star\,.\,x)\;42)$
$(\lambda y{:}\texttt{Int}.\,y)\;((\lambda x{:}\star\,.\,x)\;42)$
$(\lambda y{:}\star\,.\,y)\;((\lambda x{:}\texttt{Int}.\,x)\;42)$
$(\lambda y{:}\texttt{Bool}.\,y)\;((\lambda x{:}\star\,.\,x)\;42)$
$(\lambda y{:}\texttt{Int}.\,y)\;((\lambda x{:}\texttt{Int}.\,x)\;42)$
$(\lambda y{:}\texttt{Bool}.\,y)\;((\lambda x{:}\texttt{Bool}.\,x)\;42)$

Figure 1 A lattice of differently-annotated versions of a gradually-typed program.

1.3 The Gradual Guarantee

So far we have four criteria for gradually-typed languages, but none of them address the second informal requirement stated in the abstract: that a gradually-typed language should support the gradual migration of code between static and dynamic typing. One possible criteria is that the behavior of the program should not, roughly speaking, change due to changes in the type annotations.

For example, suppose we start with the un-annotated program at the top of the lattice in Figure 1. (In the GTLC, an un-annotated parameter is the same as annotating it with the unknown type $\star$.) One would hope that adding type annotations would yield a program that still evaluates to 42. Indeed, in the GTLC, adding the type annotation `Int` for parameters x and y does not change the outcome of the program. On the other hand, the programmer might insert the wrong annotation, say `Bool` for parameter y, and trigger a trapped error. Even worse, the programmer might add `Bool` for x and cause a static type error. So we cannot claim full contextual equivalence when going down in the lattice, but we can make a strong claim when going up in the lattice: the less precise program behaves the same as the more precise one except that it might have fewer trapped errors.

The partial order at work in Figure 1 is the natural extension of the precision relation to terms. (Type precision is also known as naive subtyping [17].) We write $T \sqsubseteq T'$ when type T is more precise than T' and $e \sqsubseteq e'$ when term e is more precisely-annotated than e'.[1] We give the definition of these relations in Section 3. We characterize the expected static and dynamic behavior of programs as we move up and down in precision as follows.

▶ **Theorem 5** (Gradual Guarantee). *Suppose $e \sqsubseteq e'$ and e is well-typed at T in the GTLC.*

1. *e' is well-typed at some T' and $T \sqsubseteq T'$.*
2. *If e evaluates to value v, then e' evaluates to some v' and $v \sqsubseteq v'$.*
   If e diverges, so does e'.
3. *If e' evaluates to value v', then e evaluates to some v where $v \sqsubseteq v'$, or to a trapped error.*
   If e' diverges, then e diverges or evaluates to a trapped error.

Now that we have stated the gradual guarantee, it is natural to wonder how important it is. Of course, there are many pressures at play during the design of any particular programming language, such as concerns for efficiency, safety, learning curve, and ease of implementation. However, if a language is intended to support gradual typing, that means the programmer should be able to conveniently migrate code from being statically typed to dynamically typed, and vice versa. With the gradual guarantee, the programmer can be confident that when removing type annotations, a well-typed program will continue to be well-typed (with no need to insert explicit casts) and a correctly-running program will continue to do so. When

[1] We apologize that the direction of increase in precision is to the left instead of to the right. We chose this directionality to be consistent with naive subtyping.

$$\begin{tikzcd}
(\lambda y{:}\star\,.\,y)\ ((\lambda x{:}\star\,.\,x)\ \texttt{ref}\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{Ref Int}.\,y)\ ((\lambda x{:}\texttt{Ref}\ \star\,.\,x)\ \texttt{ref}\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{Ref Int}.\,y)\ ((\lambda x{:}\texttt{Ref Int}.\,x)\ \texttt{ref}\ 42)
\end{tikzcd}$$

Figure 2 A lattice of varying-precision for a program with mutable references.

adding type annotations, if the program remains well typed, the only possible change in behavior is a trapped error due to a mistaken annotation. Furthermore, it is natural to consider tool support (via static or dynamic analysis) for adding type annotations, and we would not want the addition of types to cause programs to misbehave in unpredictable ways.

2 Critiques of Language Designs in Light of the Gradual Guarantee

Researchers have already explored a large number of points in the design space for gradually-typed languages. A comprehensive survey is beyond the scope of this paper, but we have selected a handful of them to discuss in light of the gradual guarantee.

2.1 Gradually-Typed Lambda Calculus (GTLC)

The Gradually-Typed Lambda Calculus [9] satisfies the gradual guarantee (see Section 3).

2.2 GTLC with Mutable References

Siek and Taha [9] treat mutable references as invariant in their type system, disallowing implicit casts that change the pointed-to type. Consider that design in relation to the lattice of programs in Figure 2. The program at the top is well-typed because `Ref Int` may be implicitly cast to $\star$ (anything can). The program at the bottom is well-typed; it contains no implicit casts. However, the program in the middle is not well-typed because one cannot cast between `Ref Int` and `Ref` $\star$. These programs are related by precision and the bottom program is well-typed, so part 1 of the gradual guarantee is violated.

Herman et al. [5, 6] remedy the situation with a design for mutable references that allows implicit casts between reference types so long as the pointed-to types are consistent (in the technical sense of Siek and Taha [9]). We conjecture that their design satisfies the gradual guarantee. Likewise, we conjecture that the new monotonic approach [10] to mutable references satisfies the gradual guarantee.

2.3 TS*

The TS* language [12] layers a static type system over JavaScript to provide protection from security vulnerabilities. TS* is billed as a gradually-typed language, but it does not satisfy the gradual guarantee. For example, consider the program variations in Figure 3. The function with parameter f is representative of a software framework and the function with parameter x is representative of a client-provided callback.

A typical scenario of gradual evolution would start with the untyped program at the top, proceed to the program in the middle, in which the framework interface has been statically typed (parameter f) but not the client, then finally evolve to the program at the bottom

$$\begin{tikzcd}
(\lambda f : \star.\, f\ \texttt{true})\ (\lambda x : \star.\, x) \arrow[d, dash] \\
(\lambda f : \texttt{Bool}{\to}\texttt{Bool}.\, f\ \texttt{true})\ (\lambda x : \star.\, x) \arrow[d, dash] \\
(\lambda f : \texttt{Bool}{\to}\texttt{Bool}.\, f\ \texttt{true})\ (\lambda x : \texttt{Bool}.\, x)
\end{tikzcd}$$

Figure 3 A lattice of varying-precision for a higher-order program.

which is fully-typed. The top-most and bottom-most versions evaluate to `true` in TS*. However, the middle program produces a trapped error, as explained by the following quote.

> "Coercions based on `setTag` can be overly conservative, particularly on higher-order values. For example, trying to coerce the identity function $id : \star \to \star$ to the type $\texttt{Bool} \to \texttt{Bool}$ using `setTag` will fail, since $\star \not<: \texttt{Bool}$." [12]

The example in Figure 3 is a counterexample to part 2 of the gradual guarantee; the bottom program evaluates to `true`, so the middle program should too, but it does not. The significance of not satisfying the gradual guarantee, as we can see in this example, is that programmers will encounter trapped errors in the process of refactoring type annotations, and will be forced to make several coordinated changes to get back to a well-behaved program.

2.4 Thorn and Like Types

The Thorn language [18] is meant to support the evolution of (dynamically-typed) scripts to (statically-typed) programs. The language provides a dynamic type **dyn**, nominal class types, and novel *like* types.

We revisit parts of Figure 1 under several scenarios. First, suppose we treat $\star$ as **dyn** and `Int` is a concrete type (i.e. a class type). Then we have the following counterexample to part 1 of the gradual guarantee; the bottom program is well-typed but not the top program.

$$\begin{tikzcd}
(\lambda y{:}\texttt{Int}.\, y)\ ((\lambda x{:}\mathbf{dyn}.\, x)\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{Int}.\, y)\ ((\lambda x{:}\texttt{Int}.\, x)\ 42)
\end{tikzcd}$$

Second, suppose again that $\star$ is **dyn** but that we replace `Bool` with `like Bool` and `Int` with `like Int`. Now every program in Figure 1 is well-typed, even the bottom-right program that should not be:

$$(\lambda y{:}\texttt{like Bool}.\, y)\ ((\lambda x{:}\texttt{like Bool}.\, x)\ 42)$$

So in this scenario the gradual guarantee is satisfied, but not the correspondence with a fully-static language (Theorem 1). Finally, suppose we replace $\star$ with `like Int` and treat `Int` as a concrete type. Similar to the first scenario, we get the following counterexample to part 1 of the gradual guarantee; the bottom program is well-typed but not the top program. (It would need an explicit cast to be well-typed.)

$$\begin{tikzcd}
(\lambda y{:}\texttt{Int}.\, y)\ ((\lambda x{:}\texttt{like Int}.\, x)\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{Int}.\, y)\ ((\lambda x{:}\texttt{Int}.\, x)\ 42)
\end{tikzcd}$$

We note that efficiency is an important design consideration for Thorn and that it is challenging to satisfy the gradual guarantee and efficiency at the same time. For example, only recently have we found a way to support mutable references in an efficient way [10].

2.5 Grace and Structural Type Tests

The Grace language [2] is gradually typed and includes a facility for pattern matching on structural types. Boyland [3] observes that, depending on the semantics of the pattern matching, the gradual guarantee may not hold for Grace. Here we consider an example in an extension of the GTLC with a facility for testing the type of a value: the expression e `is` T returns `true` if e evaluates to a value of type T and `false` otherwise. Boyland [3] considers three possible interpretations what "a value of type T" means: an optimistic semantics that checks whether the type of the value is *consistent* with the given type, a pessimistic semantics that checks whether the type of the value is *equal* to the given type, and a semantics similar in spirit to that of Ahmed et al. [1], which only checks the top-most type constructor.

Consider the following example where g is a function that tests whether its input is a function of type `Int`→`Int`. On the right we show a lattice of several programs that apply g to the identity function.

$$\begin{tikzcd}
g \equiv (\lambda f : \star.\, f \;\texttt{is}\; \texttt{Int}{\to}\texttt{Int}) & g\ (\lambda x{:}\star.\, x) \arrow[d, dash] & g\ (\lambda x{:}\star.\, x) \arrow[d, dash] \\
 & g\ (\lambda x{:}\texttt{Int}.\, x) & g\ (\lambda x{:}\texttt{Bool}.\, x)
\end{tikzcd}$$

- Under the optimistic semantics, $g\ (\lambda x{:}\star.\, x)$ and $g\ (\lambda x{:}\texttt{Int}.\, x)$ evaluate to `true` but $g\ (\lambda x{:}\texttt{Bool}.\, x)$ evaluates to `false`. So part 2 of the gradual guarantee is violated.
- Under the pessimistic semantics, $g\ (\lambda x{:}\star.\, x)$ evaluates to `false` whereas $g\ (\lambda x{:}\texttt{Int}.\, x)$ program evaluates to `true`, so this is a counterexample to part 2 of the gradual guarantee.
- Under the semantics of Ahmed et al. [1], this program is disallowed syntactically. We could instead have $g \equiv (\lambda f : \star.\, f \;\texttt{is}\; \star{\to}\star)$ and then all three programs would evaluate to `true`. We conjecture that this semantics does satisfy the gradual guarantee.

2.6 Typed Racket and the Polymorphic Blame Calculus

We continue our discussion of type tests, but expand the language under consideration to include parametric polymorphism, as found in Typed Racket [14] and the Polymorphic Blame Calculus [1]. Consider the following programs in which a test-testing function is passed to another function, either simply as $\star$ or at the universal type $\forall\alpha.\, \alpha{\to}\alpha$.

$$\begin{tikzcd}
(\lambda f : \star.\, f[\texttt{Int}]\ 5)\ (\Lambda\alpha.\, \lambda x : \star.\, x \;\texttt{is}\; \texttt{Int}) \arrow[d, dash] \\
(\lambda f : \forall\alpha.\, \alpha{\to}\alpha.\, f[\texttt{Int}]\ 5)\ (\Lambda\alpha.\, \lambda x : \star.\, x \;\texttt{is}\; \texttt{Int})
\end{tikzcd}$$

In the bottom program, 5 is sealed when it is passed to the polymorphic function f. In Typed Racket, a sealed value is never an integer, so the type test returns `false`. The program on the top evaluates to `true`, so this is a counterexample to part 2 of the gradual guarantee.

In the Polymorphic Blame Calculus, applying a type test to a sealed value always produces a trapped error, so this is not a counterexample under that design. We conjecture that one could extend the GTLC with polymorphism and compile it to the Polymorphic Blame Calculus to obtain a language that satisfies the gradual guarantee.

2.7 Reticulated Python and Object Identity

During the evaluation of Reticulated Python, a gradually-typed variant of Python, Vitousek et al. [16] encountered numerous problems when adding types to third-party Python libraries and applications. The root of these problems was a classic one: proxies interfere with object identity. (The standard approach to ensuring soundness for gradually-typed languages is to create proxies when casting higher-order entities like functions and objects.) Consider the GTLC extended with mutable references and an operator named `alias`? for testing whether two references are aliased to the same heap cell. Then in the following examples, the bottom program evaluates to `true` whereas the top program evaluates to `false`.

$$\begin{tikzcd}
\texttt{let}\, r : \texttt{ref Int} = \texttt{ref}\, 0\, \texttt{in}\ (\lambda x : \texttt{ref Int}.\, \lambda y : \texttt{ref}\ \star\, .\, \texttt{alias?}\, x\, y)\, r\, r \arrow[d, dash] \\
\texttt{let}\, r : \texttt{ref Int} = \texttt{ref}\, 0\, \texttt{in}\ (\lambda x : \texttt{ref Int}.\, \lambda y : \texttt{ref Int}.\, \texttt{alias?}\, x\, y)\, r\, r
\end{tikzcd}$$

There are several approaches to mitigate this problem, such as changing `alias`? to see through proxies, use the membrane abstraction [15], or avoid proxies altogether [18, 16, 10]. One particularly thorny issue for Reticulated Python is that the use of the foreign-function interface to C is common in Python programs and the foreign functions are privy to a rather exposed view of Python objects.

3 The Proof of the Gradual Guarantee for the GTLC

Here we summarize the proof of the gradual guarantee for the GTLC. All of the definitions, the proof of the main lemma (Lemma 7), and its dependencies, have been verified in Isabelle [7]. They are available at the following URL:
`https://dl.dropboxusercontent.com/u/10275252/gradual-guarantee-proof.zip`

For the sake of brevity, we refer to Siek and Taha [9] for the type system of the GTLC ($\Gamma \vdash_G e : T$). The dynamic semantics of the GTLC is defined by a cast-inserting type-directed translation to a cast calculus ($\Gamma \vdash e \rightsquigarrow f : T$ where f ranges over terms of the cast calculus). We use a translation similar to the one in Herman et al. [6] (ignore the mutable references) and not the one by Siek and Taha [9] because the gradual guarantee fails with their translation. For the definition of the cast calculus, including its dynamic semantics ($f \longmapsto f'$), we refer the reader to the "twosomes" of Siek and Wadler [11]. We define type and term precision in Figure 4. Here explicit casts take the form $e : T \Rightarrow^{\ell} T'$.

Part 1 of the gradual guarantee is easy to prove by induction on $e \sqsubseteq e'$. The proof of part 2 is interesting and will be the focus of our discussion. Part 3 is a corollary of part 2.

The semantics of the GTLC is defined by translation to the cast calculus, so our main lemma concerns a variant of part 2 that is adapted to the cast calculus. For this we need a notion of term precision for the cast calculus. Further, the translation to the cast calculus needs to preserve precision. Figure 4 defines precision for the cast calculus in a way that satisfies this need by adding rules that allow extra casts on both the left and right-hand side.

▶ **Lemma 6** (Cast Insertion Preserves Precision). *Suppose* $\Gamma \vdash e \rightsquigarrow f : T$, $\Gamma' \vdash e' \rightsquigarrow f' : T'$, $\Gamma \sqsubseteq \Gamma'$, *and* $e \sqsubseteq e'$. *Then* $\Gamma, \Gamma' \vdash f \sqsubseteq f'$ *and* $T \sqsubseteq T'$.

The following is the statement of the main lemma, which establishes that less-precise programs simulate more precise programs.

▶ **Lemma 7** (Simulation of More Precise Programs). *Suppose* $\emptyset, \emptyset \vdash f_1 \sqsubseteq f_1'$, $\vdash f_1 : T$, *and* $\vdash f_1' : T'$. *If* $f_1 \longmapsto f_2$, *then* $f_1' \longmapsto^* f_2'$ *and* $f_2 \sqsubseteq f_2'$ *for some* f_2'.

With the main lemma (Lemma 7) in hand, we prove part 2 of the gradual guarantee by induction on the number of reduction steps. Part 3 is a corollary of part 2, as follows.

Type Precision $\boxed{T \sqsubseteq T}$

$$\frac{}{T \sqsubseteq \star} \qquad \frac{}{B \sqsubseteq B} \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4}$$

Term Precision for the GTLC $\boxed{e \sqsubseteq e}$

$$\frac{}{k \sqsubseteq k} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{T_1 \sqsubseteq T_2 \quad e_1 \sqsubseteq e_2}{\lambda x{:}T_1.\, e_1 \sqsubseteq \lambda x{:}T_2.\, e_2} \qquad \frac{e_1 \sqsubseteq e_2 \quad e_1' \sqsubseteq e_2'}{(e_1\ e_1')^\ell \sqsubseteq (e_2\ e_2')^\ell}$$

Term Precision for the Cast Calculus $\boxed{\Gamma, \Gamma' \vdash f \sqsubseteq f'}$

$$\ldots \qquad \frac{\Gamma, \Gamma' \vdash f \sqsubseteq f' \quad T_1 \sqsubseteq T_1' \quad T_2 \sqsubseteq T_2'}{\Gamma, \Gamma' \vdash (f : T_1 \Rightarrow^\ell T_2) \sqsubseteq (f' : T_1' \Rightarrow T_2')} \qquad \frac{\Gamma' \vdash f' : T' \quad G \sqsubseteq T'}{\Gamma, \Gamma' \vdash \mathtt{blame}_G\, \ell \sqsubseteq f'}$$

$$\frac{\begin{array}{cc}\Gamma, \Gamma' \vdash f \sqsubseteq f' & \Gamma' \vdash f' : T' \\ T_1 \sqsubseteq T' & T_2 \sqsubseteq T'\end{array}}{\Gamma, \Gamma' \vdash (f : T_1 \Rightarrow^\ell T_2) \sqsubseteq f'} \qquad \frac{\begin{array}{cc}\Gamma, \Gamma' \vdash f \sqsubseteq f' & \Gamma \vdash f : T \\ T \sqsubseteq T_1' & T \sqsubseteq T_2'\end{array}}{\Gamma, \Gamma' \vdash f \sqsubseteq (f' : T_1' \Rightarrow T_2')}$$

$$\frac{\Gamma, \Gamma' \vdash v \sqsubseteq v'}{\Gamma, \Gamma' \vdash (v : G \Rightarrow \star) \sqsubseteq (v' : G \Rightarrow \star)} \qquad \frac{\Gamma, \Gamma' \vdash v \sqsubseteq v' \quad \Gamma \vdash v : T \quad T \sqsubseteq G}{\Gamma, \Gamma' \vdash v \sqsubseteq (v' : G \Rightarrow \star)}$$

Figure 4 Type and Term Precision.

Assume that e' evaluates to v'. Because e is well typed, it may either evaluate to a value v, evaluate to a trapped error, or diverge. If it evaluates to some v, then we have $v \sqsubseteq v'$ by part 2 and because reduction is deterministic. If e results in a trapped error, we are done. If e diverges, then so does e' by part 2, but that is a contradiction.

4 Conclusion

Gradual typing promises to enable the migration of programs between the dynamic and static type disciplines and it has seen considerable uptake both from industry and academia. However, these promises can be kept only so long as language designers formulate their gradually typed languages in the right way. In this paper, we emphasize the need for precise correctness criteria for gradually-typed languages and offer one such criteria, the gradual guarantee, that captures an essential aspect of the migration of code between statically and dynamically typed code. The current landscape of gradually typed languages reveals that this aspect has been either silently included or unfortunately overlooked, but never explicitly taken into consideration. We put forward the gradual guarantee as part of an ongoing research program to complete the set of correctness criteria for gradual typing. Such work intends to offer guidance for language designers and ultimately shed light to the foundational question: *what is gradual typing?*

Acknowledgments We thank Sam Tobin-Hochstadt, Philip Wadler, Andrew Kent, Ambrose Bonnaire-Sergeant, Andre Kuhlenschmidt, and Ronald Garcia for their input. The gradual guarantee was discovered independently by John Boyland and the Gradual Typing Group at Indiana University in 2014. The gradual guarantee is sketched in Boyland's paper at the FOOL workshop [3] and in Siek's presentation at the NII Shonan meeting on Contracts [8].

References

1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In *Symposium on Principles of Programming Languages*, January 2011.

2 Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: The absence of (inessential) difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 85–98, New York, NY, USA, 2012. ACM.

3 John Tang Boyland. The problem of structural type tests in a gradual-typed language. In *Foundations of Object-Oriented Langauges*, FOOL, 2014.

4 R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002.

5 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

6 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010.

7 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, November 2007.

8 Jeremy G. Siek. Design and evaluation of gradual typing for Python. `https://dl.dropboxusercontent.com/u/10275252/shonan-slides-2014.pdf`, May 2014.

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

10 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *European Symposium on Programming*, ESOP, April 2015.

11 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, POPL, pages 365–376, January 2010.

12 Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *ACM Conference on Principles of Programming Languages (POPL)*, January 2014.

13 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

14 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, January 2008.

15 Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM.

16 Michael M. Vitousek, Jeremy G. Siek, Andrew Kent, and Jim Baker. Design and evaluation of gradual typing for Python. In *Dynamic Languages Symposium*, 2014.

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

18 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages*, POPL, pages 377–388, 2010.