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 theory for 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 evolution 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 that includes a new formal property, named the *gradual guarantee*, that relates the behavior of programs that differ only with respect to 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 also 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

Statically and dynamically typed languages have complementary strengths. Static typing guarantees the absence of type errors, facilitates the generation of efficient code, and provides machine-checked documentation. On the other hand, dynamic typing enables rapid prototyping, flexible programming idioms, and fast adaptation to changing requirements. The theory of *gradual typing* provides both of these typing disciplines within a single language, puts the programmer in control of which discipline is used for each region of code, provides seamless interoperability, and enables the convenient evolution of code between the two disciplines. Gradual typing touches both the static type system and the dynamic semantics of a language. The key innovation in the static type system is the *consistency* relation on types, which allows implicit casts to and from the unknown type, here written $\star$, while still catching static type errors [5, 27, 49].[1] The dynamic semantics for gradual typing is based on the semantics

* This work was partially supported by NSF grant 1360694.
[1] The consistency relation is also known as compatibility.

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of contracts [19, 24], coercions [33], and interlanguage migration [40, 62]. Because of the shared mechanisms with these other lines of research, much of the ongoing research benefits the theory of gradual typing, and vice versa [15–17, 25, 26, 28, 39, 56].

Gradual typing has a syntactic similarity to type inference [42], which also supports optional type annotations. However, type inference differs from gradual typing in that it requires static type checking for the whole program. However, there are many other lines of research that seek to integrate static and dynamic typing, listed below.

1. *Dynamic & Typecase* extends statically typed languages with a type, often named Dynamic or Any, together with explicit forms for injecting and projecting values into and out of the Dynamic type [1, 12, 38].
2. *Object-Oriented Languages* extend statically typed languages with a type, often called Object, with implicit conversions into Object and explicit conversions out of Object [23, 37].
3. *Soft Typing* applies static analysis to dynamically typed programs for the purposes of optimization [13] and debugging [21].
4. *Type Hints* in dynamically typed languages enable type specialization in optimizing compilers, such as in Lisp [55] and Dylan [47].
5. *Types for Dynamic Languages* design static type systems for dynamically typed languages, such as StrongTalk [10] and Typed Racket [63].

While these lines of research share the broad goal of integrating statically and dynamically typed languages, they address goals different from those associated with gradual typing.

To a large degree, the practice of gradual typing preceded the theory. A number of languages provided a combination of static and dynamic type checking, with implicit casts to and from the unknown type. These languages included Cecil [14], Visual Basic.NET [41], Bigloo [46], and ProfessorJ [24]. The theory of gradual typing provides a foundation for these languages. Thatte's earlier theory of Quasi-static Typing [60] almost meets the goals of gradual typing, but it does not statically catch all type errors in completely annotated programs. (See Siek and Taha [49] for an in-depth discussion.)

Over the last decade there has been significant interest, both in academia and industry, in the integration of static and dynamic typing. On the industry side, there is Dart [59], TypeScript [6, 32], Hack [65], and the addition of Dynamic to C# [31]. On the academic side, there is a growing body of research [2–4, 6, 8, 9, 18, 22, 30, 34, 36, 40, 44, 45, 50, 52–54, 57, 58, 61, 66–69]. The term *gradual typing* is often used to describe language designs that integrate static and dynamic typing. However, some of these designs do not satisfy the original intent of gradual typing because they do not support the convenient evolution of code between the two typing disciplines. This goal was implicit in the original paper; it did not include a formal property that captures convenient evolution. To this end, we offer a new criterion in this paper, the *gradual guarantee*, that relates the behavior of programs that differ only with respect to the precision of their type annotations.

In Section 2 we discuss several example programs that demonstrate gradual typing and motivate the need for the gradual guarantee. We review the semantics of the Gradually Typed Lambda Calculus (GTLC) (Section 3) and then state the formal criteria for gradually typed languages, including the gradual guarantee (Section 4). In Section 5 we survey some of the gradual typing literature, critiquing designs in light of the gradual guarantee. The last section before the conclusion reports on a mechanized proof that the GTLC satisfies the gradual guarantee (Section 6).

2 Examples of Gradual Typing

In this section, we highlight the goals of gradual typing by way of several examples and motivate the need for the gradual guarantee. The examples are written in Reticulated Python, a gradually typed variant of Python [66] using the syntax for type annotations specified in PEP 484 [29]. For example, a function type $T_1 \to T_2$ is written `Callable[[`T_1`],`T_2`]`.

2.1 Gradual Typing Includes Both Fully Static and Fully Dynamic

The first goal of gradual typing is to provide both fully static type checking and fully dynamic type checking. In other words, a gradually typed language can be thought of being a superset of two other languages, a fully static one and a fully dynamic one. For example, the GTLC is, roughly speaking, a superset of both the Simply Typed Lambda Calculus (STLC) and the (Dynamically Typed) Lambda Calculus (DTLC). We say that a program is *fully annotated* if all variables have type annotations and if the type $\star$ does not occur in any of the type annotations. A fully annotated program of the GTLC should behave the same as in the STLC, and a program without type annotations should behave the same as in the DTLC. An important aspect of a program's behavior that we take into account is the error cases, of which there are several varieties: 1) a program may fail to type check, 2) a program may encounter a runtime error and the language definition requires that the program halt or raise an exception, i.e., a *trapped error* [11]) and 3) a program may encounter a runtime error that the language definition says nothing about, i.e., an *untrapped error*. The STLC, GTLC, and even DTLC are all strongly typed so they are free of untrapped errors. Furthermore, the STLC is free of trapped errors and so is the GTLC on fully annotated programs.

Consider the examples in Figure 1. The GCD_{1a} and GCD_{1b} functions at the top have no type annotations whereas GCD_{3a} and GCD_{3b} at the bottom are fully annotated. The un-annotated versions should behave just like they would in Python. Indeed, with GCD_{1a}, the call `gcd(15, 9)` returns `(3,-1,2)`. Can you spot the error in GCD_{1b}? The second return statement is returning a pair instead of a 3-tuple. But that error is not caught statically because the programmer has asked for dynamic checking. Turning to the fully annotated versions GCD_{3a} and GCD_{3b}, they should behave just as they would in some hypothetical statically typed variant of Python. Indeed, with GCD_{3a}, the call `gcd(15, 9)` returns `(3,-1,2)` and furthermore, the gradual type system guarantees that GCD_{3a} is free of runtime type errors. On the other hand, with GCD_{3b}, a static error is reported to indicate that returning a pair conflicts with the return type `Tuple[int,int,int]`.

2.2 Gradual Typing Provides Sound Interoperability

With gradual typing, fully static programs behave the same as if they were written in a statically typed programming language. As a result, they are guaranteed not to encounter type errors at runtime. But what about partially typed programs?

Consider the following algorithm for computing the modular inverse. We have not annotated the parameters of `modinv`, so it is dynamically typed, but suppose it calls the statically typed GCD_{3b}. What happens if someone forgets a conversion and passes a string as parameter `m` of `modinv`?

```
def modinv(a, m):
    (g, x, y) = gcd(a, m)
    if g != 1: raise Exception()
    else: return x % m
```

GCD$_{1a}$

```
def gcd(a, b):
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y, y)
```

GCD$_{1b}$

```
def gcd(a, b):
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y)
```

GCD$_{3a}$

```
def gcd(a:int, b:int)
      -> Tuple[int,int,int]:
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y, y)
```

GCD$_{3b}$

```
def gcd(a:int, b:int)
      -> Tuple[int,int,int]:
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y)
```

Figure 1 Static and dynamic variants of the extended greatest-common divisor algorithm.

Does the string flow into the `gcd` function and trigger a runtime error in the expression `b % a`? That would be unfortunate, because `gcd` is statically typed and one would hope that `gcd` is guaranteed to be free of runtime errors, of both the trapped and untrapped variety. Further, there would be a string masquerading as an integer and programmers would not be able to trust their type annotations. With gradual typing, the runtime system protects the static typing assumptions by casting values as they flow between statically and dynamically typed code. So in this example, there is a cast error in `modinv` just before the call to `gcd`. In fact, gradual typing ensures that statically typed regions of code are free of runtime type errors.

Next let us consider the following fully annotated version of `modinv` with a call to the dynamically typed GCD$_{1a}$. Because this function refers to a variable (`gcd`) that is dynamic, this function is only partially typed.

```
def modinv(a : int, m : int):
    (g, x, y) = gcd(a, m)
    if g != 1: raise Exception()
    else: return x % m
```

However, one would like to understand which parts of `modinv` are safe versus which parts might result in a runtime type error. We accomplish this by analyzing the implicit casts in `modinv`. In the call `gcd(a,m)`, the arguments are cast from `int` to `Any`. In general, gradual typing guarantees that upcasts like these are safe. (We define upcast in terms of subtyping in Section 4.2.) On the other hand, the return type of GCD$_{1a}$ is unspecified, so it defaults to `Any`. Thus, the assignment to the tuple `(g, x, y)` requires a downcast, which is unsafe. One thing worth noting is that some partially typed code can be completely safe. For example, if we changed GCD$_{1a}$ to have return type `Tuple[int,int,int]` (but leave the parameter types unspecified), then the `modinv` function would be safe because the only implicit casts would be the upcasts on the arguments in the call to `gcd`. We envision that IDE's for gradually typed languages will provide feedback to programmers, identifying the locations of unsafe casts.

Next we consider the situation in which a dynamically typed function is used as a callback

inside a statically typed function. In the following we compute the derivative of a function `fun` at two different points. However, there is an error during the second call to `deriv` because `fun` returns the `None` value when the input is not positive.

```
def deriv(d: float, f: Callable[[float],float], x: float) -> float:
  return (f(x + d) - f(x - d)) / (2.0 * d)

def fun(y):
  if y > 0: return y ** 3 - y - 1

deriv(0.01, fun, 3.0)
deriv(0.01, fun, -3.0)
```

As described above, gradual typing performs runtime casts to ensure that values are consistent with their static types. Here the cast needs to check that `fun` has type `Callable[[float], float]`. However, determining the return type of an arbitrary dynamically typed function is undecidable. (The halting problem reduces to this problem.) Instead, gradual typing draws on research for contracts [19] and delays the checking until the function is called. Thus, a cast error occurs inside `deriv` when parameter `f` is applied to a negative number.

If this were the end of the story, it would be unfortunate; as we stated above, statically typed code should be free of runtime type errors. In this example, the code that called `deriv` is at fault but the error occurs inside `deriv`. However, thanks to the blame tracking technique of Findler and Felleisen [19], the cast error can point back to the call to `deriv` and explain that `fun` violated the expected type of `Callable[[float],float]` returning `None`. Thus, in general, the soundness guarantee for gradual typing is stated in terms of blame: upcasts never result in blame, only downcasts or cross-casts.

2.3 Gradual Typing Enables Gradual Evolution

So far we demonstrated how gradual typing subsumes static and dynamic typing and provides sound interoperability between the two, but we have not demonstrated the *gradual* part of gradual typing. That is, programmers should be able to add or remove type annotations without any unexpected impacts on their program, such as whether it still typechecks and whether its runtime behavior remains the same. In Figure 2 we show several points in the evolution of the `gcd` function with respect to dynamic versus static typing. These versions of the `gcd` function have bodies identical to GCD_{3a}; they only differ in their type annotations.

One might naively want all of the versions in Figure 2 to have exactly the same behavior with respect to type checking and execution. However, GCD_{3c} has the *wrong* annotation, with a return type of `Tuple[int,int]`. To ensure type soundness, a gradual type system must reject this program as ill typed. Thus, when a programmer adds annotations, they can sometimes trigger a static type error. Similarly, adding the wrong annotation can sometimes trigger a runtime error, which is good because it make sure that annotations stay consistent with the code. For example, when using GCD_{2a} (whose second parameter has unknown type), adding `str` as the annotation for the `m` parameter of `modinv`, as shown below, does not trigger a static error, but it does trigger a cast error at the recursive call to `gcd`.

```
def modinv(a, m : str):
  (g, x, y) = gcd(a, m)
  ...
modinv(3, 'hi %s')
```

GCD$_{1a}$

```
def gcd(a, b): ...
```

GCD$_{2a}$

```
def gcd(a:int, b): ...
```

GCD$_{3a}$

```
def gcd(a:int, b:int)
       -> Tuple[int,int,int]:
  if a == 0: return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y, y)
```

GCD$_{2b}$

```
def gcd(a, b:int): ...
```

GCD$_{3c}$

```
def gcd(a:int, b:int)->Tuple[int,int]:
  ...
```

Figure 2 Evolutions of the extended greatest-common divisor algorithm.

What about the reverse? What does removing type annotations do to the behavior of a gradually typed program? The *gradual guarantee* says that if a gradually typed program is well typed, then removing type annotations always produces a program that is still well typed. Further, if a gradually typed program evaluates to a value, then removing type annotations always produces a program that evaluates to an equivalent value.

One of the primary use cases for gradual typing is to enable the evolution of programs from untyped to typed. Thus, one might be disappointed that the graduate guarantee is not as strong when moving in that direction. However, the gradual guarantee has more to say about this direction: a program remains well typed so long as only correct type annotations are added. We take *correct* to mean that the annotation agrees with the corresponding annotation in some fully annotated and well-typed version of the program. With this definition, one can apply the gradual guarantee to show that the program of interest remains well typed with the addition of correct type annotations.

3 The Gradually Typed Lambda Calculus

Here we review the GTLC [49] (Figure 3), which extends the STLC with the unknown type $\star$ and un-annotated functions. The blame labels ℓ represent source position information from the parser, so blame labels are unique. The typing rules for constants, variables, and functions are the same as in the STLC. The two key aspects of the GTLC type system can be seen in the rule for application. First, the consistency relation, written $T_1 \sim T_2$, is used in GTLC where the STLC would check for type equality. The consistency relation is more liberal when it comes to the unknown type: it relates any type to the unknown type. For example, consistency is responsible for the the call `gcd(15, 9)`, with version GCD$_{1a}$, being well typed. The argument types are `int`, the parameter types are $\star$, and `int` $\sim \star$. The consistency relation is responsible for rejecting `gcd(32, true)`, with GCD$_{3a}$, because `bool` $\not\sim$ `int`. In contrast to subtyping, consistency is symmetric but not transitive. Gradual typing can be added to object-oriented languages by combining subtyping and consistency in a principled fashion [6, 50]. Another important aspect of the GTLC is the metafunction *fun*. The GTLC allows a term in function position to be of type $T_1 \to T_2$ or of type $\star$. The *fun* metafunction extracts the domain and codomain type, treating $\star$ as if it were $\star \to \star$ [22].

Blame labels ℓ

Base types $B ::= \texttt{int} \mid \texttt{bool}$

Types $T ::= B \mid T \to T \mid \star$

Constants $k ::= \texttt{true} \mid 0 \mid \texttt{inc} \mid \cdots$

Expressions $e ::= k \mid x \mid \lambda x{:}T.\, e \mid (e\ e)^{\ell}$

$\lambda x.\, e \equiv \lambda x{:}\star .\, e$

Consistency $\boxed{T \sim T}$

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

Expression Typing $\boxed{\Gamma \vdash e : T}$

$$\cdots \quad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad \mathit{fun}(T_1) = T_{11} \to T_{12} \qquad T_2 \sim T_{11}}{\Gamma \vdash (e_1\ e_2)^{\ell} : T_{12}}$$

Function Matching $\boxed{\mathit{fun}(T) = T \to T}$

$$\mathit{fun}(T_{11} \to T_{12}) = T_{11} \to T_{12}$$
$$\mathit{fun}(\star) = \star \to \star$$

Dynamic Semantics: $e \Downarrow r \equiv \emptyset \vdash e \rightsquigarrow f : T$ and $f \longmapsto^* r$ for some f and T.

Figure 3 The Gradually Typed Lambda Calculus (GTLC).

The dynamic semantics of the GTLC is defined by translation into an internal cast calculus much like the Blame Calculus [67]. The internal cast calculus extends the STLC with the unknown type $\star$ but it replaces the implicit casts of the GTLC with explicit casts. The cast calculus and translation is defined in Figure 4. The cast expression has the form $f : T \Rightarrow^{\ell} T$, which enables a left-to-right reading. We abbreviate a sequence of two casts $(e : T_1 \Rightarrow^{\ell_1} T_2) : T_2 \Rightarrow^{\ell_2} T_3$ as follows to avoid repetition: $e : T_1 \Rightarrow^{\ell_1} T_2 \Rightarrow^{\ell_2} T_3$. The translation from the GTLC to the cast calculus is defined by the judgment $\Gamma \vdash e \rightsquigarrow f : T$. Again, application is the interesting case. We insert a cast on the expression in function position to make sure it has type $T_1 \to T_2$ and we insert a cast around the argument to make sure it has type T_1. Each inserted cast corresponds to the use of *fun* or $\sim$ in the type system.

The dynamic semantics of the cast calculus is given in Figure 4. The first things to note are the two value forms specific to casts. A value enclosed in a cast between two function types is itself a value, which we call a *wrapped* function. A value that is cast from a ground type G to $\star$ is also a value, which we call an *injection*. Ground types include only base types and the function type $\star \to \star$.[2]

Now we turn to the reduction rules for casts. Identity casts on base types and $\star$ are discarded (rule IdBase and IdStar). When an injection (a cast from a ground type to $\star$) meets a projection (a cast from $\star$ to a ground type) then either the ground types are equal and the casts are discarded (rule Succeed) or the ground types differ and the program halts and assigns blame (rule Fail). A cast to or from $\star$ that involves a non-ground type is decomposed into two casts with a ground type in the middle (rules Ground and Expand). The Ground rule is necessary to ensure that injections are restricted to ground types. The Expand rule is not strictly necessary but allows Succeed and Fail to focus on ground type.

Perhaps the most interesting rule concerns casts between function types (rule AppCast). Recall the example in Section 2.2 in which the function `fun` of type $\star \to \star$ is passed to `deriv` at type `float` $\to$ `float` and applied to 3.01. The cast from $\star \to \star$ to `float` $\to$ `float` applied to `fun` is a value. The application of a wrapped function breaks the cast into two parts: a

[2] Restricting injections to ground types gives the UD semantics [53, 67]. To instead obtain the D semantics, this restriction can be lifted [53].

$$
\begin{array}{llll}
\text{Ground Types} & G & ::= & B \mid \star\to\star \\
\text{Expressions} & f & ::= & k \mid x \mid \lambda x{:}T.\, f \mid f\ f \mid f : T \Rightarrow^{\ell} T \mid \mathtt{blame}_T\, \ell \\
\text{Values} & v & ::= & k \mid \lambda x{:}T.\, f \mid v : T_1{\to}T_2 \Rightarrow^{\ell} T_3{\to}T_4 \mid v : G \Rightarrow^{\ell} \star \\
\text{Results} & r & ::= & v \mid \mathtt{blame}_T\, \ell \\
\text{Frames} & F & ::= & \Box\ f \mid v\ \Box \mid \Box : T_1 \Rightarrow^{\ell} T_2
\end{array}
$$

Expression Typing $\boxed{\Gamma \vdash_C f : T}$

$$
\cdots \qquad \frac{\Gamma \vdash_C f : T_1 \qquad T_1 \sim T_2}{\Gamma \vdash_C (f : T_1 \Rightarrow^{\ell} T_2) : T_2} \qquad \frac{}{\Gamma \vdash_C \mathtt{blame}_T\, \ell : T}
$$

Cast Insertion $\boxed{\Gamma \vdash e \rightsquigarrow f : T}$

$$
\cdots \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow f_1 : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow f_2 : T_2 \\ \mathit{fun}(T_1) = T_{11} \to T_{12} \qquad T_2 \sim T_{11}\end{array}}{\Gamma \vdash (e_1\ e_2)^{\ell} \rightsquigarrow (f_1 : T_1 \Rightarrow^{\ell} T_{11}{\to}T_{12})\ (f_2 : T_2 \Rightarrow^{\ell} T_{11}) : T_{12}}
$$

Dynamic Semantics $\boxed{f \longmapsto f}$

$$
\begin{array}{rll}
(\lambda x{:}T.\, f)\ v \longmapsto [x := v]f & & (\textsc{Beta}) \\
v : B \Rightarrow^{\ell} B \longmapsto v & & (\textsc{IdBase}) \\
v : \star \Rightarrow^{\ell} \star \longmapsto v & & (\textsc{IdStar}) \\
v : G \Rightarrow^{\ell_1} \star \Rightarrow^{\ell_2} G \longmapsto v & & (\textsc{Succeed}) \\
v : G_1 \Rightarrow^{\ell_2} \star \Rightarrow^{\ell_2} G_2 \longmapsto \mathtt{blame}_{G_2}\, \ell_2 & \text{if } G_1 \neq G_2 & (\textsc{Fail}) \\
(v_1 : T_1{\to}T_2 \Rightarrow^{\ell} T_3{\to}T_4)\ v_2 \longmapsto v_1\ (v_2 : T_3 \Rightarrow^{\ell} T_1) : T_2 \Rightarrow^{\ell} T_4 & & (\textsc{AppCast}) \\
v : T \Rightarrow^{\ell} \star \longmapsto v : T \Rightarrow^{\ell} G \Rightarrow^{\ell} \star & \text{if } T \neq \star, T \neq G, T \sim G & (\textsc{Ground}) \\
v : \star \Rightarrow^{\ell} T \longmapsto v : \star \Rightarrow^{\ell} G \Rightarrow^{\ell} T & \text{if } T \neq \star, T \neq G, T \sim G & (\textsc{Expand}) \\
F[f] \longmapsto F[f'] & \text{if } f \longmapsto f' & (\textsc{Cong}) \\
F[\mathtt{blame}_{T_1}\, \ell] \longmapsto \mathtt{blame}_{T_2}\, \ell & \text{if } \vdash F : T_1 \Rightarrow T_2 & (\textsc{Blame})
\end{array}
$$

Figure 4 Cast insertion and the internal cast calculus.

cast on the argument and on the return value. The following shows the important steps.

$$
\begin{array}{l}
\quad (\mathtt{fun} : \star \to \star \Rightarrow^{\ell} \mathtt{float} \to \mathtt{float})\, 3.01 \\
\longmapsto (\mathtt{fun}\, (3.01 : \mathtt{float} \Rightarrow^{\ell} \star)) : \star \Rightarrow^{\ell} \mathtt{float} \\
\longmapsto^{*} 5.02 : \mathtt{float} \Rightarrow^{\ell} \star \Rightarrow^{\ell} \mathtt{float} \quad \longmapsto \quad 5.02
\end{array}
$$

Recall that the second call to `deriv` resulted in a cast error. In that case `fun` is applied to -2.99. The result of the function is a `None` value, which cannot be cast to `float`. Thanks to the blame tracking, the error $\mathtt{blame}_{\mathtt{float}}\, \ell$ includes the source information for the cast that arose from passing `fun` into `deriv`.

$$
\begin{array}{l}
\quad (\mathtt{fun} : \star \to \star \Rightarrow^{\ell} \mathtt{float} \to \mathtt{float})\ \ -2.99 \\
\longmapsto (\mathtt{fun}\, (-2.99 : \mathtt{float} \Rightarrow \star)) : \star \Rightarrow^{\ell} \mathtt{float} \\
\longmapsto^{*} \mathtt{None} : \mathtt{NoneType} \Rightarrow^{\ell} \star \Rightarrow^{\ell} \mathtt{float} \quad \longmapsto \quad \mathtt{blame}_{\mathtt{float}}\, \ell
\end{array}
$$

4 Criteria for Gradual Typing

We begin by reviewing the formal criteria for gradually typed languages that appear in the literature. These criteria cover the first three subsections of Section 2. We then develop a formal statement of our new criterion, the gradual guarantee. For the sake of precision, we state each criterion as a theorem about the GTLC. Our intent is that one would adapt these theorems to other gradually typed languages.

4.1 Gradual as a Superset of Static and Dynamic

As discussed in Section 2.1, a gradually typed language is intended to include both an untyped language and a typed language. For example, GTLC should encompasses both DTLC and the STLC. Siek and Taha [49] prove that the GTLC type system is equivalent to the STLC on fully annotated programs. We extend this criterion to require the dynamic semantics of the GTLC to coincide with the STLC on fully annotated programs in the theorem below. Let $\vdash_S$ and $\Downarrow_S$ denote the typing judgment and evaluation function of STLC, respectively. 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. *$\vdash_S e : T$ if and only if $\vdash e : T$. (Siek and Taha [49]).*
2. *$e \Downarrow_S v$ if and only if $e \Downarrow v$.*

The relationship between the GTLC and DTLC is more nuanced by necessity because there exist terms that are both fully annotated and un-annotated. Suppose the constant `inc` has type `int→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 well-typed program in the DTLC (trivially, because programs are). Similar issues arise when extending the GTLC with other constructs, such as conditionals, where one must choose to lean towards either static typing or dynamic typing. Nevertheless, there is a simple encoding of the DTLC into the GTLC, here written as $\lceil \cdot \rceil$, that casts constants to the unknown type. Let $\Downarrow_D$ denote evaluation of DTLC.

▶ **Theorem 2** (Embedding of DTLC)**.** *Suppose that e is a term of DTLC.*
1. *$\vdash \lceil e \rceil : \star$ (Siek and Taha [49]).*
2. *$e \Downarrow_D r$ if and only if $\lceil e \rceil \Downarrow \lceil r \rceil$.*

The two theorems of this section characterize programs at the two extremes: fully static and fully dynamic. However, these theorems say nothing about partially typed programs which is the norm for gradually typed languages. The next section describes notions of soundness that make sense for partially typed programs.

4.2 Soundness for Gradually Typed Languages

Siek and Taha [49] 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 could be ubiquitous). Let $e \Uparrow$ indicate that e diverges.

▶ **Theorem 3** (Type Safety of GTLC, Siek and Taha [49])**.** *If $\vdash e : T$, then either $e \Downarrow v$ and $\vdash v : T$ for some v, or $e \Downarrow \mathtt{blame}_T\, \ell$ for some ℓ, or $e \Uparrow$.*

This theorem is unsatisfying because it does not tell us that statically typed regions are not to blame for trapped errors. Thankfully, blame tracking provides the right mechanism

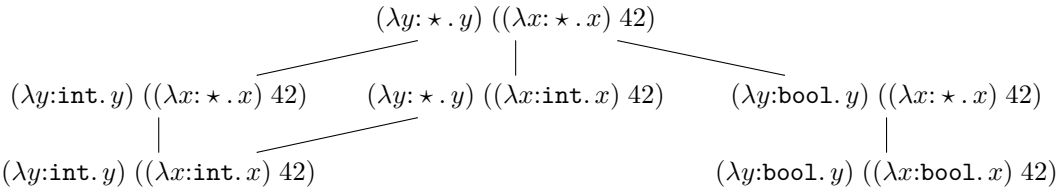

Figure 5 A lattice of differently annotated versions of a gradually typed program.

for formulating type soundness for gradually typed programs. The Blame Theorem [62, 67] characterizes the safe versus possibly unsafe casts. Adapting these results to gradual typing, we arrive at the Blame-Subtyping Theorem, which states that if the cast insertion procedure inserts a cast from a type to another and the former is a subtype of the latter, then the cast is guaranteed not to fail. To state this theorem, we give the following definition of subtyping.

$$\boxed{T <: T}$$

$$B <: B \qquad \star <: \star \qquad \frac{T <: G}{T <: \star} \qquad \frac{S_1 <: T_1 \quad T_2 <: S_2}{T_1 \to T_2 <: S_1 \to S_2}$$

▶ **Theorem 4** (Blame-Subtyping Theorem). *If* $\emptyset \vdash e \rightsquigarrow f : T$ *,* f *contains a cast* $f' : T_1 \Rightarrow^{\ell} T_2$*,* $T_1 <: T_2$*, and* $e \Downarrow \mathtt{blame}_T\, l'$ *then* $l \neq l'$*.*

The practical implication of this theorem is that a programmer (or automated tool) can easily analyze a region of code to tell whether they region is safe or whether it might trigger a cast error. Furthermore, the Blame-Subtyping Theorem guarantees that fully-static regions of code are never to blame for cast errors.

4.3 The Gradual Guarantee

So far we have four theorems to characterize gradually typed languages, but none of them address the requirement of Section 2.3. Roughly speaking, changes to the annotations of a gradually typed program should not change the static or dynamic behavior of the program.

For example, suppose we start with the un-annotated program at the top of the lattice in Figure 5. 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 5 is the *precision relation* on types and terms, defined in Figure 6. Type precision [51] is also known as naive subtyping [67]. Term precision is the natural extension of type precision to terms. Here 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'.[3] We give the definition of these relations in Section 6. We characterize the expected static and dynamic behavior of programs as we move up and down in precision as follows.

[3] We apologize that the direction of increase in precision is to the left instead of to the right. We settled on this directionality to be consistent with subtyping.

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 $\boxed{e \sqsubseteq e}$

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

Figure 6 Type and Term Precision.

▶ **Theorem 5** (Gradual Guarantee). *Suppose $e \sqsubseteq e'$ and $\vdash e : T$.*

1. *$\vdash e' : T'$ and $T \sqsubseteq T'$.*
2. *If $e \Downarrow v$, then $e' \Downarrow v'$ and $v \sqsubseteq v'$.*
   If $e \Uparrow$ then $e' \Uparrow$.
3. *If $e' \Downarrow v'$, then $e \Downarrow v$ where $v \sqsubseteq v'$, or $e \Downarrow \mathtt{blame}_T\, l$.*
   If $e' \Uparrow$, then $e \Uparrow$ or $e \Downarrow \mathtt{blame}_T\, l$.

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 evolve 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 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.

5 Critiques of Language Designs in Light of the Gradual Guarantee

Researchers have 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.

5.1 GTLC

As discussed above, the Gradually Typed Lambda Calculus [49] satisfies the gradual guarantee (the proof is in Section 6).

5.2 GTLC with Mutable References

Siek and Taha [49] 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 7. 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

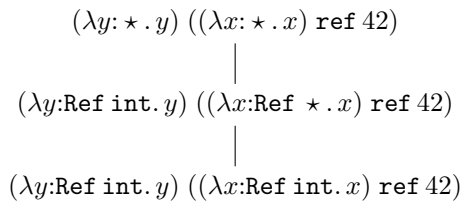

Figure 7 A lattice of varying-precision for a program with mutable references.

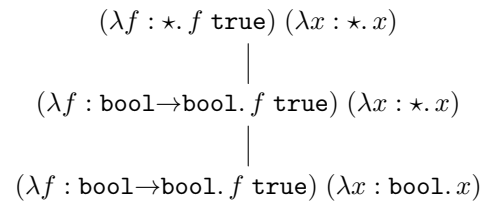

Figure 8 A lattice of varying-precision for a higher-order program.

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. [34, 35] 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 [49]). We conjecture that their design satisfies the gradual guarantee. Likewise, we conjecture that the new monotonic approach [54] to mutable references satisfies the gradual guarantee.

5.3 TS*

The TS* language [57] 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 8. 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 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 `bool` $\to$ `bool` using `setTag` will fail, since $\star \not<:$ `bool`." [57]

The example in Figure 8 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.

5.4 Thorn and Like Types

The Thorn language [69] 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 *like* types.

We revisit parts of Figure 5 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.

$$\begin{array}{c}(\lambda y{:}\texttt{int}.\, y)\ ((\lambda x{:}\textbf{dyn}.\, x)\ 42)\\ \big|\\ (\lambda y{:}\texttt{int}.\, y)\ ((\lambda x{:}\texttt{int}.\, x)\ 42)\end{array}$$

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 5 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.)

$$\begin{array}{c}(\lambda y{:}\texttt{int}.\, y)\ ((\lambda x{:}\texttt{like int}.\, x)\ 42)\\ \big|\\ (\lambda y{:}\texttt{int}.\, y)\ ((\lambda x{:}\texttt{int}.\, x)\ 42)\end{array}$$

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 without using wrappers [54].

5.5 Grace and Structural Type Tests

The Grace language [7] is gradually typed and includes a facility for pattern matching on structural types. Boyland [9] 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 [9] 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. [2], 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.

$$g \equiv (\lambda f : \star.\, f\ \texttt{is int}{\to}\texttt{int})$$

$$\begin{array}{c}g\ (\lambda x{:}\star\,.\, x)\\ \big|\\ g\ (\lambda x{:}\texttt{int}.\, x)\end{array} \qquad \begin{array}{c}g\ (\lambda x{:}\star\,.\, x)\\ \big|\\ g\ (\lambda x{:}\texttt{bool}.\, x)\end{array}$$

- 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. [2], 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.

5.6 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 [28, 63] and the Polymorphic Blame Calculus [2]. Consider the following programs in which a test-testing function is passed to another function, either simply as $\star$ or at the universal type $\forall\alpha.\,\alpha{\to}\alpha$.

$$(\lambda f : \star.\, f[\texttt{int}]\ 5)\ (\Lambda\alpha.\,\lambda x : \star.\, x\ \texttt{is int})$$
$$\Big|$$
$$(\lambda f : \forall\alpha.\,\alpha{\to}\alpha.\, f[\texttt{int}]\ 5)\ (\Lambda\alpha.\,\lambda x : \star.\, x\ \texttt{is int})$$

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.

5.7 Reticulated Python and Object Identity

During the evaluation of Reticulated Python, a gradually typed variant of Python, Vitousek et al. [66] 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 [20]. (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`.

$$\texttt{let}\ r : \texttt{ref int} = \texttt{ref}\ 0\ \texttt{in}\ \ (\lambda x : \texttt{ref int}.\, \lambda y : \texttt{ref}\ \star\ .\, \texttt{alias?}\, x\, y)\ r\ r$$
$$\Big|$$
$$\texttt{let}\ r : \texttt{ref int} = \texttt{ref}\ 0\ \texttt{in}\ \ (\lambda x : \texttt{ref int}.\, \lambda y : \texttt{ref int}.\, \texttt{alias?}\, x\, y)\ r\ r$$

There are several approaches to mitigate this problem, such as changing `alias`? to see through proxies, use the membrane abstraction [64], or avoid proxies altogether [54, 66, 69]. 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.

6 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 [43]. They are available at the following URL:
`https://dl.dropboxusercontent.com/u/10275252/gradual-guarantee-proof.zip`
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.

Because the semantics of the GTLC is defined by translation to the cast calculus, our main lemma concerns a variant of part 2 that is adapted to the cast calculus. For this we

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

$$\cdots \quad \frac{\Gamma, \Gamma' \vdash f \sqsubseteq f' \quad T_1 \sqsubseteq T_1' \quad T_2 \sqsubseteq T_2'}{\Gamma, \Gamma' \vdash (f : T_1 \Rightarrow^{\ell_1} T_2) \sqsubseteq (f' : T_1' \Rightarrow^{\ell_2} T_2')} \qquad \frac{\Gamma' \vdash f' : T' \quad T \sqsubseteq T'}{\Gamma, \Gamma' \vdash \mathtt{blame}_T\, \ell \sqsubseteq f'}$$

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

Abbreviation: $f \sqsubseteq f' \equiv \emptyset, \emptyset \vdash f \sqsubseteq f'$

Figure 9 Term Precision for the Cast Calculus.

need a notion of precision for the cast calculus. Further, the translation to the cast calculus needs to preserve precision. Figure 9 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* $T \sqsubseteq T'$.

This lemma is interesting in that it was, in fact, not true for the original formulation of the GTLC [49]. In that version, there were two cast insertion rules for function application, one where the term in function position had an arrow type and one where it had type $\star$. The latter case used the following rule.

$$\frac{\Gamma \vdash e_1 \rightsquigarrow f_1 : \star \quad \Gamma \vdash e_2 \rightsquigarrow f_2 : T}{\Gamma \vdash (e_1\ e_2) \rightsquigarrow ((f_1 : \star \Rightarrow T \to \star)\ f_2) : \star}$$

Using this rule, if we take $e_1 = (((\lambda g{:}\star \to \star.\, g)\ (\lambda x{:}\star\,.\, x))\ 42)$ and $e_2 = (((\lambda g{:}\star\,.\, g)\ (\lambda x{:}\star\,.\, x))\ 42)$, we have that $e_1 \sqsubseteq e_2$. However, when we obtain f_1, f_2 by $\emptyset \vdash e_1 \rightsquigarrow f_1 : \star$ and $\emptyset \vdash e_2 \rightsquigarrow f_2 : \star$, we get

$$f_1 = (((\lambda g{:}\star \to \star.\, g)\ (\lambda x{:}\star\,.\, x))\ (42 : \mathtt{Int} \Rightarrow \star))$$
$$f_2 = ((((\lambda g{:}\star\,.\, g)((\lambda x{:}\star\,.\, x){:}\star \to \star \Rightarrow \star)){:}\star \Rightarrow \mathtt{Int} \to \star)\ 42)$$

and $f_1 \not\sqsubseteq f_2$.

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* $f_1 \sqsubseteq f_1'$, $\vdash f_1 : T$, *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'.

The proof of the Lemma 7 is by induction on the derivation of $f_1 \sqsubseteq f_1'$ followed by case analysis on $f_1 \longmapsto f_2$. The proof required four major lemmas (and numerous minor lemmas).

Because the precision relation of the cast calculus allows extra casts on the right-hand side, we prove the following lemma by case analysis on T_1' and T_2'. The precision relation also allows extra casts on the left, but we handled those cases in-line in the proof of Lemma 7.

▶ **Lemma 8** (Extra Cast on the Right). *Suppose* $\vdash_C v : T$, $\vdash_C v' : T_1'$, $T \sqsubseteq T_1'$, *and* $T \sqsubseteq T_2'$. *If* $v \sqsubseteq v'$, *then* $v' : T_1' \Rightarrow^{\ell} T_2' \longmapsto^* v''$ *and* $v \sqsubseteq v''$ *for some* v''.

To handle cases where the more-precise program is already a value, we prove that the less-precise program can reduce to a related value. This proof is by induction on the derivation of $v \sqsubseteq f'$.

▶ **Lemma 9** (Catchup to Value on the Left). *Suppose* $\vdash_C v : T$, $\vdash_C f' : T'$, *and* $v \sqsubseteq f'$. *Then* $f' \longmapsto^* v'$ *and* $v \sqsubseteq v'$.

The most complex part of the proof involved application, as functions may be wrapped in a series of casts. We prove the following lemma by induction on the derivation of $(\lambda x : T_1.\, f) \sqsubseteq v_1'$.

▶ **Lemma 10** (Simulation of Function Application). *Suppose* $\vdash_C (\lambda x : T_1.\, f) : T_1 \to T_2$, $\vdash_C v : T_1$, $\vdash_C v' : T_1$, $\vdash_C v_1' : T_1' \to T_2'$, $\vdash_C v_2' : T_1'$, *and* $T_1 \to T_2 \sqsubseteq T_1' \to T_2'$. *If* $(\lambda x : T_1.\, f) \sqsubseteq v_1'$ *and* $v \sqsubseteq v_2'$, *then* $v_1'\ v_2' \longmapsto^* f'$, $[x := v]f \sqsubseteq f'$, *and* $\vdash_C f' : T_2'$.

We prove the next lemma by induction on $(v_1 : T_1 \to T_2 \Rightarrow^\ell T_3 \to T_4) \sqsubseteq v_1'$.

▶ **Lemma 11** (Simulation of Unwrapping). *Suppose* $\vdash_C v_1 : T_1 \to T_2$, $\vdash_C v_2 : T_1$, $\vdash_C v_1' : T_1' \to T_2'$, $\vdash_C v_2' : T_1'$, *and* $T_1 \to T_2 \sqsubseteq T_1' \to T_2'$. *If* $(v_1 : T_1 \to T_2 \Rightarrow^\ell T_3 \to T_4) \sqsubseteq v_1'$ *and* $v_2 \sqsubseteq v_2'$, *then* $v_1'\ v_2' \longmapsto^* f'$ *and* $v_1\ (v_2 : T_3 \Rightarrow^\ell T_1) : T_2 \Rightarrow^\ell T_4 \sqsubseteq f'$.

With 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. 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.

7 Conclusion

Gradual typing should allow programmers to straightforwardly evolve their programs between the dynamic and static typing disciplines. However, this is only available to the programmer if the language designer formulates their language in a specific way. In this paper, we emphasize the need for formal criteria for gradually typed languages and offer a new criterion, the gradual guarantee. This formal property captures essential aspects of the evolution of code between typing disciplines. 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. That we could formally prove that GTLC obeys the gradual guarantee is a promising step and indicates that it is a realistic goal for researchers designing gradually typed systems. It remains to be investigated whether the gradual guarantee can be proven for a full-blown language with modern features (such as polymorphism, recursive types, type inference, etc.). We look forward to working with the research community to address this challenge.

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 [9] and in Siek's presentation at the NII Shonan meeting on Contracts [48].

References

1 Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

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

3 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, August 2013.

4 Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 251–270, New York, NY, USA, 2014. ACM.

5 Christopher Anderson and Sophia Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82. Elsevier, 2003.

6 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2014.

7 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.

8 Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the jvm. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 117–136, 2009.

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

10 Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.

11 Luca Cardelli. *Handbook of Computer Science and Engineering*, chapter Type Systems. CRC Press, 1997.

12 Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Not.*, 27(8):15–42, August 1992.

13 Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

14 Craig Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.

15 Olaf Chitil. Practical typed lazy contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 67–76, New York, NY, USA, 2012. ACM.

16 Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 215–226, New York, NY, USA, 2011. ACM.

17 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, 2012.

18 Tim Disney and Cormac Flanagan. Gradual information flow typing. In *Workshop on Script to Program Evolution*, 2011.

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

20 R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming*, 2004.

21 Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999.

22 Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Symposium on Principles of Programming Languages*, POPL, pages 303–315, 2015. 10.1145/2676726.2676992. URL `http://doi.acm.org/10.1145/2676726.2676992`.

23 James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Sun Developer Network, 1996.

24 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.

25 Michael Greenberg. Space-efficient manifest contracts. In *POPL*, 2015.

26 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL) 2010*, 2010.

27 Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

28 Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium*, 2007.

29 Łukasz Langa Guido van Rossum, Jukka Lehtosalo. Type hints. `https://www.python.org/dev/peps/pep-0484/`, September 2014. draft.

30 Lars T. Hansen. Evolutionary programming and gradual typing in ECMAScript 4 (tutorial). Technical report, ECMA TG1 working group, November 2007.

31 Anders Hejlsberg. C# 4.0 and beyond by anders hejlsberg. Microsoft Channel 9 Blog, April 2010.

32 Anders Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, 2012.

33 Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

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

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

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

37 International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. September 1998.

38 Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU reference manual. Technical Report LCS-TR-225, MIT, October 1979.

39 Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *Proceedings of the 17th European Symposium on Programming (ESOP'08)*, March 2008.

40 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007.

41 Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

42 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

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

44 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *Symposium on Principles of Programming Languages*, POPL, pages 481–494, January 2012.

45 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. Technical Report MSR-TR-2014-99, Microsoft Research, 2014.

46 Manuel Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.

47 Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

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

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

50 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, volume 4609 of *LCNS*, pages 2–27, August 2007.

51 Jeremy G. Siek and Manish Vachharajani. Gradual typing and unification-based inference. In *DLS*, 2008.

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

53 Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, ESOP, pages 17–31, March 2009.

54 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.

55 Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

56 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, 2012.

57 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.

58 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 793–810, 2012.

59 The Dart Team. *Dart Programming Language Specification*. Google, 1.2 edition, March 2014.

60 Satish Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press.

61 Peter Thiemann and Luminous Fennell. Gradual typing for annotated type systems. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 47–66. Springer Berlin Heidelberg, 2014.

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

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

64 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.

65 Julien Verlaguet. Facebook: Analyzing PHP statically. In *Commercial Users of Functional Programming (CUFP)*, 2013.

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

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

68 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, ECOOP'11. Springer-Verlag, 2011.

69 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.