Confined Gradual Typing *

Esteban Allende[1] Johan Fabry[1] Ronald Garcia[2] Éric Tanter[1]

[1]PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Chile
[2]Software Practices Laboratory, Computer Science Department, University of British Columbia, Canada
{eallende,jfabry,etanter}@dcc.uchile.cl, rxg@cs.ubc.ca

Abstract

Gradual typing combines static and dynamic typing flexibly and safely in a single programming language. To do so, gradually typed languages implicitly insert casts where needed, to ensure at runtime that typing assumptions are not violated by untyped code. However, the implicit nature of cast insertion, especially on higher-order values, can jeopardize reliability and efficiency: higher-order casts can fail at any time, and are costly to execute. We propose Confined Gradual Typing, which extends gradual typing with two new type qualifiers that let programmers control the flow of values between the typed and the untyped worlds, and thereby trade some flexibility for more reliability and performance. We formally develop two variants of Confined Gradual Typing that capture different flexibility/guarantee tradeoffs. We report on the implementation of Confined Gradual Typing in Gradualtalk, a gradually-typed Smalltalk, which confirms the performance advantage of avoiding unwanted higher-order casts and the low overhead of the approach.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords gradual typing; performance; reliability

1. Introduction

Combining static and dynamic typing is attracting a lot of attention, both from industry (*e.g.* TypeScript, Dart) and academia (*e.g.* [1, 4–7, 10, 13, 14, 17, 19, 20, 24, 25, 28]). Gradual typing [19, 20] is a partial typing technique that allows developers to define which sections of code are statically typed and which are dynamically typed, at a very fine level of granularity, by selectively placing type annotations where desired. The type system ensures that untyped code does not violate the assumptions made in statically-typed code. This makes it possible to choose between the flexibility of dynamic typing and the reliability of static typing.

The semantics and implementation of a gradually-typed language typically proceed by translation to an intermediate language with casts, *i.e.* runtime type checks that control the boundaries between typed and untyped code. Casts are key to realizing the flexibility of gradual typing. However, these casts impact programs on two fronts: reliability and efficiency. First, reliability is affected because casts may fail at runtime. In particular, when higher-order values cross the typed/untyped boundary, runtime checks may be delayed, and may eventually fail within the context of typed code. Effectively, this means that the boundaries between typed and untyped code are dynamic, and hence hard to reason about and predict, especially when integrating components from different parties. Second, efficiency can be compromised if higher-order casts are executed repeatedly.

While the flexibility provided by gradual typing is certainly a strong asset, reliability and efficiency are not to be taken lightly. The problem is that existing gradually-typed languages allow any value to cross the typed/untyped boundaries. As a result, the programmer has no direct control over which values may be passed across boundaries, making it hard to predict the resulting behavior. For instance, missing type annotations and untyped third-party library can have unexpected consequences. Of course, programming in a gradually-typed language means embracing the possibility of runtime errors. But it is not necessary to give up the possibility of ensuring that certain typed components never go into the wild, or at least do so in a controlled manner.

To address this, we develop an extension of gradual typing that adds another axis of control, so that programmers can explicitly adjust the tradeoff between flexibility and pre-

* Esteban Allende is funded by a CONICYT-Chile Ph.D. Scholarship. This work is partially funded by FONDECYT Project 1110051.

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660222

dictability. Confined Gradual Typing (CGT) refines a gradual type system with type qualifiers that restrict the flow of values between the typed and untyped parts of a program. We develop two variants of CGT: *i)* a strict variant that provides strong reliability and efficiency guarantees at the expense of some rigidity; *ii)* a relaxed variant that defers some checking to runtime, but still preserves interesting guarantees. We develop both the theory and practice of Confined Gradual Typing, using Gradualtalk [1] as a practical testbed.

This paper is structured as follows: Section 2 introduces the necessary background on gradual typing, illustrates the issues associated with the implicitness of gradual typing, and informally describes Confined Gradual Typing in its two variants. Sections 3 and 4 formalize Strict Confined Gradual Typing and Relaxed Confined Gradual Typing respectively, establishing the key properties of each approach. Section 5 briefly describes the implementation of both variants in Gradualtalk. Section 6 reports on performance measurements of CGT in Gradualtalk, highlighting the incurred cost of higher-order casts. Section 7 reviews related work, and Section 8 concludes.

Most proofs associated with the formalization are included in appendices; the complete version of all proofs can be found in the companion technical report [3]. The implementation and benchmark code are available online.[1]

2. Motivation

In this section we first give an overview of gradual typing. We then present two concrete examples that motivate the need to control the reliability and efficiency impact of gradual typing. We end this section by informally introducing Confined Gradual Typing.

2.1 Gradual Typing in a Nutshell

Gradual typing [19, 20] allows the smooth integration of static and dynamic typing by swapping the conservative pessimism of a static type system (*i.e. reject all programs that may go wrong*) for a healthy dose of optimism (*i.e. accept all programs that may go right*).

Suppose an untyped function id, and two variables x1 and x2, statically typed as String. The programmer can reasonably expect x2 = id x1 to work fine, but this is just a belief; id might return any value it wants, not necessarily a String. The gradual type system statically accepts that expectation, because the unknown type, denoted Dyn, is *consistent* [19] with any type. The consistency relation is the key for gradual typing to statically rule out programs that will clearly go wrong while being otherwise optimistic.

But the optimism of the gradual type system is no blind faith: at runtime, a check is performed to ensure that the value returned by id is indeed a String, so that the static assumption that x2 is of type String is not violated. Internally, the above program is rewritten to an intermediate representation in which *casts* are inserted:

```
x2 = <String ⇐ Dyn> id (<Dyn ⇐ String> x1)
```

The first cast from String to Dyn serves no purpose in a language whose runtime is based on tagged values. In a language with untagged values, it tags the value referred to by x1 with its type, String. When id returns, the value is cast from Dyn to String. This cast fails if id does not return a String value.

Higher-order casts. Higher-order casts are much more subtle. Consider a variation of the above example, in which x1 is typed as String → String and x2 has type X. The intermediate representation with casts is now as follows:

```
x2 = <X ⇐ Dyn> id (<Dyn ⇐ String → String> x1)
```

If X is not a function type, and assuming id is the identity function, then a cast error is raised at runtime. If X is a function type, say A → B, then we need to know if it makes sense to treat a String → String function as a function of type A → B. This depends on whether the two function types are consistent with one another.

If they are inconsistent, *e.g.* X=Int→Int, a cast error is raised immediately. If they are consistent, however, we need to ensure that the untyped function (the result of the application of id) properly *behaves* as an A → B function. This is not decidable in general, so the runtime system must generate a *function wrapper*: a function of the expected type that internally inserts casts to the arguments and returned value of the underlying function.

To illustrate, let us call v the underlying function bound to x1. If id is the identity, it returns the tagged value <Dyn ⇐ String→String>v. The function wrapper is hence a new function of type A → B that internally applies v, with corresponding casts applied to the argument and result:

```
x2 = λ x: A. <B ⇐ String>(v <String ⇐ A>x)
```

Note that if A=B=String, then the corresponding wrapper would always succeed trivially. Therefore the wrapper can be avoided altogether, in which case x2 gets bound to v without any intervening wrapper. On the other hand, if the function cast is consistent but not equal, for instance X=Dyn→String, then the wrapper is needed to cast the argument x from Dyn to String.

For the purpose of this work, it is crucial to highlight the difference between a function value crossing the typed-untyped boundaries, and a function value being wrapped: a wrapper is not necessarily created as a result of boundary crossing. As it turns out, function wrappers are the source of several issues, both in terms of reliability and efficiency, as illustrated below.

[1] http://pleiad.cl/gradualtalk/cgt

2.2 Gradualtalk in a Nutshell

The examples below are formulated in Gradualtalk[2], a gradually-typed dialect of Smalltalk [1]. A brief introduction follows.

In Smalltalk, Number≫to: stop do: aBlock denotes the to:do: method of the Number class, which takes two parameters: stop and aBlock. Local variables are declared between pipe characters at the beginning of methods. Smalltalk closures, also called blocks, are defined in square brackets with a pipe character separating the parameters from the body, *e.g.*, 2 to: 5 do: [:i | i printOn: outputStream]

In Gradualtalk, parameter and return types are optionally specified by prefixing the parameter names and the method name with a type in parentheses. For example:

```
Number≫(Self) to: (Number) stop do: (Number → ?) aBlock
```

The types of block parameters and local variables are specified similarly. Self denotes the type of the receiver and ? is the unknown type (missing type annotations are interpreted as the unknown type). Note the use of function-like syntax for block types, even though blocks are objects. Gradualtalk supports nominal types, structural types (not used in this paper), and generics, among others [1]. Type variables are given using lowercase letters, *e.g.* Collection<e> or (a → a).

2.3 Reliability

We now describe an example where the flexibility of gradual typing produces a reliability issue that is hard to track down.

The application. Consider the construction of a large, data-intensive application written in Gradualtalk. One team is in charge of the statistics functionality. The Stats object is responsible for running statistics on a subset of the data, which it keeps as a typed instance variable data of type GTCollection<Number>. The GTCollection generic class, provided by Gradualtalk, is part of a new collection hierarchy that is fully typed. Of interest here is the inject:into: method—the Smalltalk equivalent of a left fold—whose type signature is:

```
GTCollection<e> ≫ (a) inject: (a)aVal into: (a e → a)aBlock
```

The block that computes the statistics is kept as an instance variable of Stats. The accessors of this instance variable are typed, *e.g.* as follows:

```
Stats ≫ (Self) statBlock: (Integer Integer → Integer)aBlock
 statBlock := aBlock
```

However, due to an oversight, the instance variable statBlock itself is left untyped (we do not show its declaration here). Lastly, the following method runs the statistics:

```
Stats ≫ (Integer) basicStats
 ^self data inject: 0 into: self statBlock
```

[2] http://pleiad.cl/gradualtalk

A separate development team is responsible for the user interface. A UIStats class allows the user to choose which statistic is calculated by using a dropbox widget.

```
UIStats ≫ (Self) setStat: (Symbol)statName
  self stats statBlock: (self statBlocks at: statName)

UIStats ≫ (Self) getStats
  self showStatResult: (self stats basicStats)
```

When the statistic to run is selected, the method setStat: is called, setting the corresponding block in the Stats object. To calculate the statistics, the user presses a button, which invokes getStats and displays the result to the user.

The problem. The careful reader will have noticed that the data collection is declared to contain Number objects, while the statBlock: setter expects a function that manipulates Integers. Because we are using a gradually-typed language, this mismatch raises no static type error. Indeed, the statBlock instance variable was left untyped, so when the argument block is assigned, it silently crosses the boundary to the untyped world. When the statistics block is used, in the body of basicStats, the gradual type system implicitly casts it back to the type Integer Number → Integer. As long as the contents of the data collection of Stats are integers, the implicit cast succeeds and goes unnoticed. For instance, the UI team can test the application with a block of type Integer Integer → Integer that sums all elements in data:

```
[:(Integer)sum :(Integer)next | sum + next]
```

However, suppose a floating point number occurs in the data set. When the statistics are run, a cast exception is raised, halting the application. The cause of the exception is that the statistics collection block expects an integer argument, but receives a float. While this certainly points to the fact that there was a float in the dataset, it does not pinpoint the source of the problem.[3] The underlying problem is that the statistics library was intended to be fully typed, yet an accidentally missing type annotation opened a reliability hole. The UI team was hoping that passing a well-typed value (the statistics block) to a typed library (the Stats object) would never cause a runtime type error. Existing gradually-typed languages do not support such a guarantee, because all values can implicitly cross to the untyped world.

2.4 Efficiency

We now turn to an example that describes the efficiency impact of gradual typing.

The application. We consider a refinement of the above example: the statistics are scheduled to run asynchronously in a delayed manner, instead of running interactively. The

[3] With first-order values, casts can always be evaluated fully at the boundaries, so the error messages are clear, and there is no need to use costly wrappers. Also, blame tracking [9, 15, 26, 27] addresses *traceability*—by reporting the source of a runtime cast error—not *reliability*—the absence of runtime cast errors.

scheduling functionality is realized by an external Scheduler library whose code is untyped. The code below shows how a statistics run is scheduled:

```
Stats ≫ (Self) scheduleBasicStatsIn: (Scheduler)scheduler when:
      (Time)time
 scheduler schedule: [:rcv :arg| rcv inject: 0 into: arg] on: self data
    with: self statBlock when: time
```

The body of the method specifies that when it is time, the scheduler should perform the fold operation specified in the first argument on the collection given as a second argument. This uses the same data and statistics block as before.

The problem. Surprisingly, the efficiency of the system is greatly affected in this setting. The reason for this is that the statistics block, which is typed, is passed to untyped code—as an argument of the scheduler method—and back to typed code—when the scheduler runs the job. Because there is a mismatch between the original type and the target type, this travel through untyped code forces the creation of a wrapper. As a result, the wrapper code (which never fails if the data only contains integers) is executed every time the block is called. This produces a non-negligible overhead, especially if the block is used frequently and its computational content is brief. As we show in Section 6.1, a slowdown of up to 10x is incurred. Worse still, because the slowdown is caused by an external library (the scheduler) that is untyped, there is no way to avoid this slowdown, apart from reimplementing or typing the scheduler library.

Identifying the source of the slowdown is furthermore not trivial, because cast insertion and wrapper creations are implicit in gradual typing (Section 2.1). We were faced with exactly this problem when performing benchmarks as part of the validation of our work on cast insertion strategies [2], and wished we had a way to predict and control where wrappers are introduced.

2.5 Confined Gradual Typing

As we have seen, in a gradually-typed language the flexibility provided by gradual typing can easily backfire and compromise reliability and efficiency. To address this, we propose Confined Gradual Typing as a means to control the implicitness of gradual typing.

The issues presented in the previous sections boil down to data flow issues: when higher-order values cross boundaries between statically- and dynamically-typed portions of a program, casts cannot be performed immediately, so wrappers are needed. Wrappers are expensive, and delay the detection of runtime type errors. In essence, Confined Gradual Typing refines a gradual type system with annotations that allow programmers to explicitly prohibit certain boundary crossings. This paper presents two flavors of CGT:

1. Strict Confined Gradual Typing (SCGT), which is resolved entirely statically, provides strong guarantees with respect to reliability and performance, but can be too restrictive at times.
2. Relaxed Confined Gradual Typing (RCGT), which defers some checks to runtime, is more flexible but has weaker static reliability guarantees than SCGT.

In both versions, two type qualifiers are introduced: $\uparrow$ and $\downarrow$. Intuitively, $\uparrow$ protects the *future* flow of a typed value, while $\downarrow$ constrains the *past* flow of a typed value. Their precise meaning differs, however, between variants, as discussed below.

2.5.1 Strict Confined Gradual Typing

In Strict Confined Gradual Typing, the $\uparrow$ qualifier, as in $\uparrow T$, expresses that an expression has type T and, once reduced to a value, it cannot flow into the untyped world. This ensures that a typed value is used in a fully typed context and hence immune to cast errors (Section 2.3).[4]

For instance, if the statistics block from the UI team is typed as $\uparrow$(Integer Integer $\rightarrow$ Integer), assigning it to an untyped instance variable is a static type error. More precisely, it cannot be passed as argument to the statBlock: method of the Stats object, unless that method *also* qualifies the type of the argument block with $\uparrow$. If that is the case, then the assignment in the body of statBlock: is a static type error, pointing to the source of the issue—the untyped instance variable.

The $\downarrow$ qualifier, as in $\downarrow T$, expresses that an expression is of type T and that its value has never flowed through the dynamic world. For a higher-order value, this ensures that the value is not wrapped, thereby avoiding performance issues (Section 2.4). For instance, the developer of the typed collection library can provide a second fold operation inject:intoSafe: where the argument block is typed with $\downarrow$:

```
GTCollection<e> ≫(a)inject: (a)aVal intoSafe: ↓(a e → a)aBlock
```

This means that aBlock should have never passed through dynamically typed code. If in the scheduler code of Section 2.4 this operation is called instead of the more permissively typed inject:into:, then the message send inject:intoSafe: is ill-typed. This is because the block passed through the dynamic world and hence is potentially wrapped.

2.5.2 Relaxed Confined Gradual Typing

Strict Confined Gradual Typing is effective in restoring predictability, but can be limiting in practice, because it systematically prohibits interaction with untyped code. Relaxed Confined Gradual Typing is a softer variant that trades the fully static guarantees of SCGT for more flexibility, while preserving interesting reliability and efficiency guarantees. Instead of focusing on whether values have flowed or will flow through untyped code, RCGT focuses on whether values have been wrapped or may be wrapped in the future.

In RCGT, the $\uparrow$ qualifier, as in $\uparrow T$, expresses the constraint that the (higher-order) value of type T will not be

[4] Of course, a programmer could explicitly wrap the function to get a dynamic version, but then that is a conscious and manifestly visible decision.

wrapped. The ↓ qualifier, in turn, expresses that a (higher-order) value has not been wrapped. RCGT statically allows qualified values to pass through untyped code, but finds a fault at runtime if wrappers are introduced. To support RCGT, the runtime system must therefore be able to recognize when a typed value is passed to untyped code and then projected out to the same type (or a supertype), hence avoiding wrapping.

In the scheduler example, this means that a block typed ↓(Integer Integer → Integer) can be passed to the untyped scheduler and then passed to the typed collection method inject:intoSafe:, because no wrapping is necessary—hence performance is unaffected. However, if the block was expected to have a different return type, wrapping would be necessary, and a runtime error would be raised to prevent the implicit creation of the wrapper.

To further illustrate the difference between SCGT and RCGT, consider a typed function f of type F, the dynamically-typed identity function id of type Dyn → Dyn, and the following program, statically legal in both SCGT and RCGT:

```
x: F' = id f
```

If we protect f with the ↑ qualifier, id f does not type check anymore in SCGT, effectively protecting f from crossing the typed/untyped boundary. On the other hand, the program does type check in RCGT: f is allowed to flow into the untyped world, as long as no wrapper is created when it flows back to the typed world. So at runtime, if F' is not the same as the declared type of f, an error is raised to prevent the illegal wrapper creation.

Compared to SCGT, RCGT is more permissive and accepts more programs. With respect to guarantees, RCGT introduces a new kind of runtime errors that denote unwanted wrapper creations. Arguably, this still improves reliability compared to standard gradual typing because wrapper creations are avoided and hence there are no delayed cast errors latent in wrappers. Put differently, these errors are raised more eagerly than in gradual typing. In the above example, if no illegal wrapper exception was raised, it means x is unwrapped and as such cannot be the cause of future cast errors. On the efficiency side, RCGT makes it possible to predict and control the implicit overhead of wrappers.

2.6 Usage Scenarios of Confined Gradual Typing

The flow qualifiers introduced by Confined Gradual Typing can be helpful to programmers following different possible methodologies. We envision three such approaches:

post-hoc. The programmer uses gradual typing without qualifiers and, when facing either a reliability or efficiency issue, she introduces qualifiers to track down the sources of these issues. Note that, compared to a debugging tool, this approach has the advantage that once the source of the problem is identified, the programmer can leave the qualifiers in place, thereby ensuring that the issue will not reappear later. The programmer can also build on the experience to introduce preventive qualifiers in other places where similar issues could appear.

upfront/provider. When developing a library that has critical components (either performance- or reliability-wise), the programmer eagerly adds qualifiers in the interface to make clear that the intention is to get static/unwrapped arguments, hinting at the fact that these qualified arguments play key roles in the overall behavior of the library. Performance-wise, examples include event callbacks in GUI components, like mouse-over or repaint, which can be called intensively. Reliability-wise, examples include error logging and exception handling code, where one wants to avoid cast errors that eclipse the underlying error, or essential system components, for instance the implementation of the gradual type checker itself.

upfront/client. A programmer develops an application that imports a fully statically-typed library, which is used in a critical manner (either performance- or reliability-wise). The programmer can defensively use qualifiers on all the callbacks of the application to make sure they do not accidentally cross the static/dynamic boundary and then flow in the imported library, compromising performance or reliability. The programmer only removes qualifiers if a specific boundary crossing is deemed harmless.

Note that the last usage scenario suggests a language design, dual to the one we formulate in this paper, in which the language is by default fully statically typed, and programmers have to explicitly introduce qualifiers that allow the introduction of dynamic checking and boundary crossing.

3. Strict Confined Gradual Typing

To make the description of Confined Gradual Typing precise, we now formalize CGT, starting with the strict variant (Section 2.5.1). We defer the presentation of the relaxed variant (Section 2.5.2) to Section 4.

To ensure that CGT can be understood and applied regardless of the considered host language, this formalization is independent of Smalltalk, and follows the approach of Siek and Taha [19], which builds upon the simply-typed lambda-calculus. We respect the pay-as-you-go motto of gradual typing, in that the runtime semantics of the language does not assume tagged values: casts to the unknown type are used to maintain source type information of untyped values only when required.[5]

The semantics of the source language is given by a type-directed translation to an internal language that makes runtime checks explicit by inserting casts. The syntax and static semantics of the source language is described in Section 3.1.

[5] The impact of using a host language whose runtime uses tagged values, as in any safe dynamically-typed language like Smalltalk, is discussed in Section 5.

Section 3.2 presents the internal language, and the translation is explained in Section 3.3.

3.1 Source Language

Syntax. We start with a lambda-calculus with base types B (for simplicity we support numbers and addition), the unknown type Dyn, and the type qualifiers $\uparrow$ and $\downarrow$ of CGT (Figure 1).

$$\begin{array}{rcll} e & ::= & n \mid \lambda x : T.e \mid e\ e \mid e + e & \text{Expressions} \\ P & ::= & B \mid \text{Dyn} & \text{Primitive Type} \\ T & ::= & P \mid T \to T \mid \uparrow T \mid \downarrow T & \text{Type} \end{array}$$

Note that we consider T up to the following equations:

$$\uparrow\uparrow T = \uparrow T \qquad \downarrow\downarrow T = \downarrow T \qquad \uparrow\downarrow T = \downarrow\uparrow T$$
$$\uparrow\text{Dyn} = \downarrow\text{Dyn} = \text{Dyn}$$

Figure 1. SCGT: Source language syntax

Subtyping. The $\uparrow$ and $\downarrow$ qualifiers induce a natural subtyping relation between types (Figure 2). Since $\downarrow$ is a guarantee about the past of a value, it is possible to lose it (SS-losedown), but not to gain it. Conversely, one can see a value of type T as a $\uparrow T$ (SS-gainup), because this adds a guarantee about the future usage of the value; $\uparrow$ cannot be lost. Also, subtyping propagates below qualifiers (SS-up, SS-down). The other rules are standard.

Directed consistency. The essence of gradual typing lies in the *consistency* relation [19], which expresses the compatibility between typed and untyped expressions. Every base type is consistent with Dyn, and function types are consistent if their constituents are consistent. Consistency is typically symmetric and never transitive, otherwise any type would be consistent with any type through consistency with Dyn.

Because Confined Gradual Typing expresses constraints on the flow of values with respect to the unknown type, we introduce a non-symmetric variant of consistency, called *directed consistency* (Figure 3). Like consistency, directed consistency is reflexive and non-transitive. The loss of symmetry is due to the qualifiers $\uparrow$ and $\downarrow$; when not present, the relation is symmetric. Unqualified base types are consistent with Dyn and vice versa (DC-rdyn, DC-ldyn). Unqualified function types can only be consistent with Dyn, and vice-versa, if the type is consistent with Dyn $\to$ Dyn, and vice versa (DC-dynfun1, DC-dynfun2). Note that these last rules are new as such in the consistency relation of a gradually-typed language; they reflect a restriction needed to preserve the guarantees of qualifiers. The non-symmetry of the relation is used to guarantee that a $\downarrow T$ value has never passed through Dyn (DC-losedown), and that a $\uparrow T$ value will never pass to Dyn (DC-gainup), but not the reverse. Finally, directed consistency subsumes subtyping (DC-sub). This implies that $\downarrow T \rightsquigarrow \uparrow T$, for every type T.

$$\begin{array}{rcll} e & ::= & \ldots \mid \langle T \Leftarrow T \rangle\ e \mid \text{CastError} & \text{Expressions} \\ f & ::= & \lambda x : T.e & \text{Function values} \\ & \mid & \langle F \Leftarrow F' \rangle f \quad \text{if } F' \not<: F & \\ b & ::= & n \mid f & \text{Base values} \\ v & ::= & b \mid \langle \text{Dyn} \Leftarrow T \rangle b \quad \text{if } T \neq \text{Dyn} & \text{Values} \\ F & ::= & T \to T \mid \downarrow F \mid \uparrow F & \text{Function type} \\ E & ::= & \Box\ e \mid v\ \Box \mid \Box + e \mid v + \Box & \text{Evaluation Frames} \\ & \mid & \langle T \Leftarrow T \rangle\ \Box & \end{array}$$

Figure 7. SCGT: Internal language syntax

Typing. Equipped with directed consistency, we can now describe the typing rules of SCGT (Figure 4). Literal values are typed with the $\downarrow$ qualifier to express that they have not (yet) passed through Dyn. (T-var) is standard.

(T-app1) corresponds to the case where the function expression is of the unknown type. In contrast to standard gradual typing [19], it is not sufficient for e_2 to be well-typed: it must also be consistent with Dyn. This may not be the case in CGT, as $\uparrow T \rightsquigarrow$ Dyn never holds.

If the function expression is typed, then it ought to be a function type (T-app2). The type of the argument expression must be consistent with the argument type of the function. Since we do not care whether the function type is qualified, the rule uses an auxiliary operator $|\cdot|$ to remove qualifiers:

$$\begin{array}{rcl} |\uparrow T| & = & |T| \\ |\downarrow T| & = & |T| \\ |T| & = & T \text{ otherwise} \end{array}$$

Finally, for addition, the sub-expressions must be consistent with Int (T-add). We write $\uparrow$Int because addition consumes the number without any further casts. The newly-produced number is qualified with $\downarrow$.

3.2 Internal Language

Syntax. Figure 7 presents the syntax of the internal language, which extends the source language syntax with casts $\langle T \Leftarrow T' \rangle$ and runtime cast errors. A value v can be either a base value b or a *tagged value* $\langle \text{Dyn} \Leftarrow T \rangle b$ (with $T \neq$ Dyn). A tagged value is a value (born typed) that was passed to Dyn. The cast to Dyn plays the role of a type tag, keeping the information that the underlying value is of type T. A base value is either a number or a function value, which is either a plain function, or a *function wrapper* $\langle F \Leftarrow F' \rangle f$. Function wrappers are interesting because they embed higher-order casts, which are the source of the problems that CGT addresses. The syntactic category F is specific to function types. Note that we require $F' \not<: F$ for $\langle F \Leftarrow F' \rangle f$ to be considered a value, because otherwise the cast is eliminated by reduction (and the wrapper avoided). Evaluation frames E are expressions with holes, and are single-frame analogues to evaluation contexts from reduction semantics.

$$\text{(SS-reflex)}\ \frac{}{T <: T} \qquad \text{(SS-trans)}\ \frac{T_1 <: T_2 \qquad T_2 <: T_3}{T_1 <: T_3} \qquad \text{(SS-fun)}\ \frac{T_3 <: T_1 \qquad T_2 <: T_4}{T_1 \to T_2 <: T_3 \to T_4}$$

$$\text{(SS-losedown)}\ \frac{}{\downarrow T <: T} \qquad \text{(SS-gainup)}\ \frac{}{T <: \uparrow T} \qquad \text{(SS-up)}\ \frac{T_1 <: T_2}{\uparrow T_1 <: \uparrow T_2} \qquad \text{(SS-down)}\ \frac{T_1 <: T_2}{\downarrow T_1 <: \downarrow T_2}$$

Figure 2. SCGT: Static subtyping

$$\text{(DC-rdyn)}\ \frac{}{B \rightsquigarrow \text{Dyn}} \qquad \text{(DC-ldyn)}\ \frac{}{\text{Dyn} \rightsquigarrow B}$$

$$\text{(DC-dynfun1)}\ \frac{T_1 \to T_2 \rightsquigarrow \text{Dyn} \to \text{Dyn}}{T_1 \to T_2 \rightsquigarrow \text{Dyn}} \qquad \text{(DC-dynfun2)}\ \frac{\text{Dyn} \to \text{Dyn} \rightsquigarrow T_1 \to T_2}{\text{Dyn} \rightsquigarrow T_1 \to T_2} \qquad \text{(DC-fun)}\ \frac{T_3 \rightsquigarrow T_1 \qquad T_2 \rightsquigarrow T_4}{T_1 \to T_2 \rightsquigarrow T_3 \to T_4}$$

$$\text{(DC-losedown)}\ \frac{T_1 \rightsquigarrow T_2}{\downarrow T_1 \rightsquigarrow T_2} \qquad \text{(DC-gainup)}\ \frac{T_1 \rightsquigarrow T_2}{T_1 \rightsquigarrow \uparrow T_2} \qquad \text{(DC-sub)}\ \frac{T_1 <: T_2}{T_1 \rightsquigarrow T_2}$$

Figure 3. SCGT: Directed consistency

$$\text{(T-num)}\ \frac{}{\Gamma \vdash n : \downarrow\text{Int}} \qquad \text{(T-abs)}\ \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1.e : \downarrow(T_1 \to T_2)} \qquad \text{(T-var)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\text{(T-app1)}\ \frac{\Gamma \vdash e_1 : \text{Dyn} \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 \rightsquigarrow \text{Dyn}}{\Gamma \vdash e_1\ e_2 : \text{Dyn}}$$

$$\text{(T-app2)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad |T_1| = T_{11} \to T_{12} \qquad T_2 \rightsquigarrow T_{11}}{\Gamma \vdash e_1\ e_2 : T_{12}}$$

$$\text{(T-add)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 \rightsquigarrow \uparrow\text{Int} \qquad T_2 \rightsquigarrow \uparrow\text{Int}}{\Gamma \vdash e_1 + e_2 : \downarrow\text{Int}}$$

Figure 4. SCGT: Typing

$$\text{(IT-num)}\ \frac{}{\Gamma \vdash n : \downarrow\text{Int}} \qquad \text{(IT-var)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(IT-abs)}\ \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1.e : \downarrow(T_1 \to T_2)} \qquad \text{(IT-err)}\ \frac{}{\Gamma \vdash \text{CastError} : T}$$

$$\text{(IT-app)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad |T_1| = T_{11} \to T_{12} \qquad T_2 <: T_{12}}{\Gamma \vdash e_1\ e_2 : T_{12}}$$

$$\text{(IT-add)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 <: \uparrow\text{Int} \qquad T_2 <: \uparrow\text{Int}}{\Gamma \vdash e_1 + e_2 : \downarrow\text{Int}} \qquad \text{(IT-cast)}\ \frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2 \qquad T_2 \rightsquigarrow T_3}{\Gamma \vdash \langle T_3 \Leftarrow T_2 \rangle e : T_3}$$

Figure 5. SCGT: Internal language typing

Typing. The typing rules of the internal language are straightforward (Figure 5). Most instances of directed consistency from the source language are replaced with static subtyping. This is because necessary instances of consistency in the source language translate to casts in the intermediate language, as described below (Section 3.3). The new rule (IT-cast) expresses that a cast is valid only if the source type T_2 is a supertype of the actual type T_1, and is consistent with the target type T_3.

Dynamic semantics. The evaluation rules of the internal language are also standard, except for a few key points. Rule (E-merge) compresses two successive casts that go through Dyn if the source type T_1 is compatible with the outer target type T_2. Note that in doing so, it strips qualifiers from T_1 to reflect that the value has gone through Dyn. If T_1 is not compatible with T_2, a runtime CastError is raised (E-merge-err). Cast errors propagate outward (E-err). (E-fcastinv) describes the application of a function wrapper to a value. Regardless

$$\text{(E-congr)}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \text{(E-app)}\ \frac{}{(\lambda x : T.e)\ v \longrightarrow e[v/x]} \qquad \text{(E-add)}\ \frac{n_3 = n_1 + n_2}{n_1 + n_2 \longrightarrow n_3}$$

$$\text{(E-merge)}\ \frac{T_1 \rightsquigarrow T_2 \qquad T_1 \neq \text{Dyn}}{\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle v \longrightarrow \langle T_2 \Leftarrow |T_1|\rangle v} \qquad \text{(E-merge-err)}\ \frac{T_1 \not\rightsquigarrow T_2}{\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle v \longrightarrow \text{CastError}}$$

$$\text{(E-err)}\ \frac{}{E[\text{CastError}] \longrightarrow \text{CastError}} \qquad \text{(E-remove)}\ \frac{T_1 <: T_2}{\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow v}$$

$$\text{(E-fcastinv)}\ \frac{|F| = T_1 \to T_2 \qquad |F'| = T_1' \to T_2' \qquad F \rightsquigarrow F' \qquad F \not<: F'}{(\langle F' \Leftarrow F\rangle\ f)\ v \longrightarrow \langle T_2' \Leftarrow T_2\rangle(f\ (\langle T_1 \Leftarrow T_1'\rangle\ v))}$$

Figure 6. SCGT: Internal language dynamic semantics

$$\text{(C-num)}\ \frac{}{\Gamma \vdash n \Rightarrow n : \downarrow\text{Int}} \qquad \text{(C-var)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x \Rightarrow x : T} \qquad \text{(C-abs)}\ \frac{\Gamma, x : T_1 \vdash e \Rightarrow e' : T_2}{\Gamma \vdash \lambda x : T_1.e \Rightarrow \lambda x : T_1.e' : \downarrow(T_1 \to T_2)}$$

$$\text{(C-app1)}\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : \text{Dyn} \qquad \Gamma \vdash e_2 \Rightarrow e_2' : T_2 \qquad T_2 \rightsquigarrow \text{Dyn}}{\Gamma \vdash e_1\ e_2 \Rightarrow (\langle (T_2 \to \text{Dyn}) \Leftarrow \text{Dyn}\rangle e_1')e_2' : \text{Dyn}}$$

$$\text{(C-app2)}\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : T_1 \qquad \Gamma \vdash e_2 \Rightarrow e_2' : T_2 \qquad |T_1| = T_{11} \to T_{12} \qquad T_2 \rightsquigarrow T_{11}}{\Gamma \vdash e_1\ e_2 \Rightarrow e_1'(\langle\!\langle T_{11} \Leftarrow T_2\rangle\!\rangle e_2') : T_{12}}$$

$$\text{(C-add)}\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : T_1 \qquad \Gamma \vdash e_2 \Rightarrow e_2' : T_2 \qquad T_1 \rightsquigarrow \uparrow\text{Int} \qquad T_2 \rightsquigarrow \uparrow\text{Int}}{\Gamma \vdash e_1 + e_2 \Rightarrow (\langle\!\langle \uparrow\text{Int} \Leftarrow T_1\rangle\!\rangle e_1') + (\langle\!\langle \uparrow\text{Int} \Leftarrow T_2\rangle\!\rangle e_2') : \downarrow\text{Int}}$$

Figure 8. SCGT: Cast insertion

of the qualifiers of F and F', it decomposes to cast the argument and result of the application.

3.3 Translating Source Programs to the Internal Language

A source language program is translated to an internal language program through cast insertion (Figure 8). Each typing rule of the source language (Figure 4) has a corresponding cast insertion rule. Whenever directed consistency is needed, the output program uses explicit casts to express the corresponding runtime checks.

We use the $\langle\!\langle \cdot \rangle\!\rangle$ operator to introduce casts only when necessary (*i.e.* when directed consistency holds but static subtyping does not):

$$\langle\!\langle T_2 \Leftarrow T_1\rangle\!\rangle e = \begin{cases} e & \text{if } T_1 <: T_2, \\ \langle T_2 \Leftarrow T_1\rangle e & \text{otherwise.} \end{cases}$$

When the operator expression of an application has unknown type, it is cast to a function that accepts the argument type (C-app1). The side condition $T_2 \rightsquigarrow \text{Dyn}$ ensures that it can use an argument of type T_2 (in order to respect the $\uparrow$ qualifier). In case the function expression is typed, a cast may be inserted for the argument expression (C-app2). (C-add) is similar.

3.4 Type Safety and Correctness of Qualifiers

Type safety of the internal language is established in a standard manner via progress and preservation:

Theorem 1. *(Progress) If $\emptyset \vdash e : T$, then e is a value, or $e =$ CastError or $\exists e'$, $e \longrightarrow e'$.*

Proof. By induction on the typing rules for e [3]. □

Theorem 2. *(Preservation) If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T'$ and $T' <: T$.*

Proof. By induction on the evaluation rules [3]. □

Also, the cast insertion translation preserves typing:

Theorem 3. *(Cast insertion preserves typing)*
If $\Gamma \vdash e \Rightarrow e' : T$ in the source language, then $\Gamma \vdash e' : T$ in the internal language.

Proof. By induction on the cast insertion rules [3]. □

However, type safety is just a safety net that does not express the essence of the guarantees that the qualifiers $\uparrow$ and $\downarrow$ are supposed to bring. We really want to prove that the $\uparrow$ qualifier ensures that a value will not pass through the Dyn type, and conversely that the $\downarrow$ qualifier ensures that a value has not passed through the Dyn type.

The proof technique we use consists of formulating a variant of the semantics of SCGT where values are marked.[6] A value is *tainted*, denoted $v^\bullet$, if it has passed through Dyn, otherwise it is *untainted*, denoted v°. A value can additionally be marked as *untaintable*, denoted $\widehat{v}$. Intuitively, values are born untainted, and are tainted whenever a pair of casts through Dyn is merged (as in rule E-merge). A value is marked untaintable when it is passed as parameter of a function whose argument type has the $\uparrow$ qualifier, or when it is cast to an $\uparrow$-qualified type. The full static and dynamic semantics of the taint-tracking language are given in Appendix A.

Once we establish type safety for the taint-tracking semantics, the main theorems are the following:

Theorem 4. *($\downarrow$ correctness) If $\emptyset \vdash v : \downarrow T$, then $v = v^\circ$.*

That is, a value of type $\downarrow T$ is necessarily untainted.

Theorem 5. *($\uparrow$ correctness) If $\emptyset \vdash \langle \text{Dyn} \Leftarrow T \rangle\ v : \text{Dyn}$, then $v \neq \widehat{v}$.*

That is, a tagged value cannot be untaintable. Together with a lemma that establishes that an untaintable value must have an $\uparrow$-qualified type (Appendix A), this expresses that the stated guarantee of $\uparrow$ is correctly maintained by the semantics.

Proof. Both theorems directly follow from the Canonical Forms lemma of the taint-tracking semantics (Appendix A). □

Finally, we establish that the taint-tracking semantics is faithful to the semantics presented in this section by defining a taint erasure function $\mathit{erase}(e)$ that takes a term e of the taint-tracking language to a term of the original language by removing the taint, and proving the following result:

Theorem 6. *(Tainting faithfulness)*
If $e \longrightarrow e'$, then $\mathit{erase}(e) \longrightarrow \mathit{erase}(e')$.

Proof. By induction on the evaluation rules of the taint-tracking semantics [3]. □

4. Relaxed Confined Gradual Typing

We now present a variant of Confined Gradual Typing that is more flexible than SCGT. Relaxed Confined Gradual Typing guarantees that costly function wrappers are not created unwantedly. In RCGT, the $\downarrow$ qualifier indicates that a function value has not been wrapped, although it may have crossed the typed/untyped boundaries. Similarly, the $\uparrow$ qualifier imposes that a function value will not be wrapped, although it is allowed to cross typed/untyped boundaries.

The overall semantic framework for RCGT is similar to that of SCGT: the source language syntax is the same, but its semantics reinterpret the meaning of the type qualifiers and loosens the constraints on directed consistency (Section 4.1). Its semantics is given by translation (unchanged) to an internal language with casts, for which only one evaluation rule is different (Section 4.2). The metatheory is fairly different (Section 4.3) however, because the guarantees implied by the qualifiers are different: $\uparrow$ and $\downarrow$ do not express guarantees about passing through the Dyn type, but instead express guarantees about function wrappers.

In RCGT, the $\uparrow$ and $\downarrow$ qualifiers are meaningless for base types, since they are not subject to function wrappers, so we impose an additional equation on the syntax of types (Figure 1): $\uparrow P = \downarrow P = P$

4.1 Directed Consistency, Revisited

$$\text{(DC-rdyn-R)}\ \frac{}{T \rightsquigarrow \text{Dyn}} \qquad \text{(DC-ldyn-R)}\ \frac{}{\text{Dyn} \rightsquigarrow T}$$

Figure 9. RCGT: Modified directed consistency. Rules (DC-dynfun1, DC-dynfun2) are removed, all other rules are preserved.

In SCGT, directed consistency plays two roles: first, it ensures that the $\downarrow$ qualifier cannot be forged, and that the $\uparrow$ qualifier cannot be lost; second, it prevents certain types from being consistent with the unknown type Dyn and vice versa. In RCGT, directed consistency is more permissive (Figure 9): only the first role is preserved; any type is consistent with Dyn (DC-rdyn-R, DC-ldyn-R). As a result, RCGT statically rejects fewer programs.

4.2 Dynamic Semantics, Revisited

All the evaluation rules of the internal language are preserved as is, except for (E-merge), which is replaced by (E-merge-R), shown in Figure 10. The only difference is that the qualifiers of the source type are not removed, because in RCGT, going through Dyn is irrelevant; what matters are function wrappers.

$$\text{(E-merge-R)}\ \frac{T_1 \rightsquigarrow T_2}{\langle T_2 \Leftarrow \text{Dyn} \rangle \langle \text{Dyn} \Leftarrow T_1 \rangle v \longrightarrow \langle T_2 \Leftarrow T_1 \rangle v}$$

Figure 10. RCGT: Modified dynamic semantics. All other rules are preserved.

While the rules are very similar, their behavior is quite different. In particular, there are new instances of (E-merge-err) that amount to runtime checking that wrappers do not get created.

For instance consider a typed function $f : F$, the dynamically-typed identity function, $id = \lambda x.x : \text{Dyn} \to \text{Dyn}$, and the program $let\ x : F' = id\ f$ (we assume F and F' are unqualified types). The program is statically legal in both SCGT and RCGT.

Now consider that we protect f with the $\uparrow$ qualifier: $f : \uparrow F$. In SCGT $id\ f$ does not type check anymore, because

[6] The idea of using additional syntax to track the flow of values and be able to use syntactic proofs was inspired by Syntactic Type Abstraction [11].

$\uparrow F \not\rightsquigarrow$ Dyn. This effectively protects f from crossing the typed/untyped boundary. On the other hand, the program does type check in RCGT: f is allowed to flow into the untyped world, as long as no wrapper is created when it flows back to the typed world. At runtime, when id returns and the value is about to be assigned to x, the composed cast $\langle F' \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow \uparrow F\rangle f$ is produced. By definition, $\uparrow F \not\rightsquigarrow F'$—the $\uparrow$ qualifier cannot be lost. So (E-merge-err) applies, resulting in a CastError that manifests that the wrapper creation is illegal.

4.3 Type Safety and Correctness of Qualifiers

Progress and preservation also hold for Relaxed Confined Gradual Typing; similarly, cast insertion preserves typing [3]. Again, the most interesting result is not type safety, but a notion of correctness for the $\uparrow$ and $\downarrow$ qualifiers. In RCGT, the qualifiers provide guarantees with respect to function wrappers. Both can be expressed in the following theorem, which states that no function wrapper has an $\uparrow$-qualified type as source type or a $\downarrow$-qualified type as target type:

Theorem 7. *(No wrapping with qualifiers)*
If $e = \langle F_2 \Leftarrow F_1\rangle$ f *is a value, then* $F_1 \neq \uparrow F_1'$ *and* $F_2 \neq \downarrow F_2'$

Proof. We first prove two lemmas that relate directed consistency from/to qualified types with static subtyping, namely: $T_1 \rightsquigarrow \downarrow T_2 \Rightarrow T_1 <: \downarrow T_2$ and $\uparrow T_1 \rightsquigarrow T_2 \Rightarrow \uparrow T_1 <: T_2$. Then, the proof proceeds by contradiction, exploiting the definition of a wrapper value, *i.e.* $T_1 \not<: T_2$ (Appendix B). □

5. Implementation

We have implemented both variants of Confined Gradual Typing, SCGT and RCGT, as an extension of Gradualtalk [1], a gradually-typed dialect of Smalltalk. The implementation can be configured to operate in one of three modes: original gradual typing (GT hereafter), SCGT and RCGT. This section briefly describes how to go from the theory of Confined Gradual Typing to its implementation in Gradualtalk.

5.1 From Theory to Practice

Objects. One of the biggest differences between the formal presentation of Confined Gradual Typing and the actual implementation is that Gradualtalk is an object-oriented language, with subtyping. Gradualtalk is built on *consistent subtyping* [20], a relation that combines consistency with traditional object subtyping. We first extend subtyping in Gradualtalk to include the subtyping rules for qualifiers (Figure 2). Second, we make the consistent subtyping relation *directed* (Figure 3). Third, because Smalltalk provides many primitive operations, we require that the arguments to the primitives be consistent subtypes of $\uparrow T$ instead of T (similar to the case of addition in the theory).

Casts to Dyn and type safety. Following seminal work on gradual typing, the formalization of CGT assumes an unsafe runtime system and relies on casts to Dyn to tag values only when necessary. Because Smalltalk is a safe dynamic language, its runtime already maintains a type tag for each value. In implementing CGT in Gradualtalk, therefore, we do not need to re-tag all values: we can discharge casts from their safety-bearing role. The only exception is closures, because we must keep track of the semantics of qualifiers, as discussed below in Section 5.2.

Type system features. Gradualtalk supports several type system features beyond nominal and function types, *e.g.* structural types and union types. As of now, the implementation of SCGT and RCGT only supports nominal and function types. Studying and implementing the semantics of qualifiers for the other features is future work.

Live system. Nearly every Smalltalk environment is a live system: the developer writes the code, runs it and debugs it all in the same execution environment. To support this live environment, class definitions can change at runtime, and individual methods can be added to and removed from an existing class. Gradualtalk already deals with this incremental and dynamic setting [1], and CGT does not introduce new challenges in this regard.

5.2 Confined Gradual Typing in Gradualtalk

In both SCGT and RCGT all syntactically created values automatically have $\downarrow$ as part of their type. Furthermore the subtype rules SS-gainup, SS-losedown, SS-up, and SS-down (Figure 2) establish how $\downarrow$ and $\uparrow$ are passed along, produced and consumed, and guarantee that $\downarrow$ cannot be forged and $\uparrow$ cannot be lost. In contrast, the default gradual typing semantics (GT) simply ignores qualifiers. SCGT has the exact same runtime as GT: only the compile-time type checker differs, raising errors related to misused qualifiers. RCGT has different static and dynamic semantics from GT. Its type checker is less strict than that of SCGT, and its runtime semantics differ from both GT and SCGT, because function casts and the type tags of closures are managed differently. We now briefly expand on the differences in RCGT.

Tagged closures. A tagged closure is an extended Smalltalk closure that contain an extra type tag. The tag is used to mark the closure with its source type when it is cast to Dyn, and to mark when it acquires the $\uparrow$ property (*i.e.* it cannot be wrapped). Tagged closures are instances of a subclass of the class of all blocks, BlockClosure. To tag an existing closure, a new tagged closure is created and all the instance variables of the original block are copied, along with the source type when casting to Dyn, or the target type when casting to an $\uparrow$-qualified type.[7]

[7] Just like the implementation of function wrappers, this implementation is vulnerable to reflective operations.

Function casts. To avoid repeating subtype tests, the implementation of function casts in Gradualtalk is a union of the E-remove, E-merge-R and E-merge-err rules (Figures 6 and 10):

1. If $\uparrow$ is present in the source type but not the target type a cast error is raised.
2. If the source type is not a subtype of the target type:
   (a) Throw a cast error if the source type has a $\uparrow$.
   (b) Throw a cast error if the target type has a $\downarrow$.
   (c) Wrap the closure object.
   (d) If the target type has an $\uparrow$, produce a tagged closure.
3. If the source type is a subtype of the target type:
   (a) If the target type has an $\uparrow$, produce a tagged closure.
   (b) Otherwise return the value unaltered.

6. Performance Evaluation

An important goal of Relaxed Confined Gradual Typing is to obtain higher performance. This gain results from the ability to avoid unwanted implicit wrapping of closures and consequently avoid their overhead on closure application. This of course supposes that wrappers have a noticeable overhead. To validate this hypothesis, we have performed some microbenchmarks and macrobenchmarks. The microbenchmarks establish the cost of boundary crossing from static to dynamic and back and, more importantly, the cost of applying wrappers. The macrobenchmarks focus on the cost of applying wrappers and see if the observations on microbenchmarks scale up to a more realistic scenario.

Both microbenchmarks and macrobenchmarks were run on a machine with an Intel Core i7 3.20 GHz CPU, 4 GB RAM and 250 GB SATA drive, running Windows 7. The VM used is Pharo.exe build number 14776 and base image is Pharo 2.0 build number 20628.

6.1 Microbenchmarks

We report on microbenchmarks that evaluate the overhead of function wrappers in Gradualtalk and the impact of RCGT at runtime. The microbenchmarks reuse the setting from Section 2. The first set of benchmarks determines the cost of a closure crossing the boundary back into typed code, while the second set determines the cost of applying wrappers. The closure used is the simple statistics block from Section 2.3, with type Integer Integer $\rightarrow$ Integer, used to perform a left fold on a collection of Number values.

Multiple runs of the benchmarks have been performed, with differing amounts of closure creation and application: from 100,000 to one million, in increments of 100,000. For each iteration count, we take the average of 10 runs. Here we only include the detailed results for one millon iterations. The detailed data shows that the observed relative performance is similar on all iteration counts (Appendix C).

6.1.1 The Cost of Boundary Crossing

We evaluate the cost of boundary crossing by benchmarking three variants of a small program that assigns a typed closure to a typed variable, after going through an untyped variable. The basic template is as follows:

```
|block|
block := [:(Integer)acc :(Integer)i|acc + i].
[ 1 to: self iterations do: [:i|
    |(Integer Integer → Integer)tblock|
    tblock := block.
    tblock class. "Cheap--prevents elimination of the loop body"
 ]] timeToRun.
```

The typed closure is first assigned to the untyped local variable block. Then the benchmark loop is defined; it is run and measured in the last line. The number of iterations in the loop is determined in the fourth line. The iteration body assigns block to a typed variable tblock, and then performs a cheap operation on that variable (basically, an instance variable access), which is necessary to prevent the compiler from optimizing away the entire body of the loop. The variants we consider are:

- **Fully Typed**: To establish a base line, we benchmark a fully-typed version of the program, so that there is no boundary crossing at all. This means that the block local variable is typed as Integer Integer $\rightarrow$ Integer.
- **No Wrapping**: This variant is the one presented in the code snippet above: the blocked is assigned to a typed variable tblock of the same type as the source type of the block; therefore, there is no need to create a wrapper.
- **Wrapping**: This variant changes the declaration of tblock to be of type Number Number $\rightarrow$ Number. Because this is not a subtype of the original type of the closure, a wrapper is created.

Note that for Fully Typed and No Wrapping, we also benchmark versions with an $\uparrow$ qualifier added at the creation of the block, or a $\downarrow$ qualifier on the type of tblock. This allows us to examine the assumption that adding qualifiers should not introduce a noticeable penalty. Of course, because type qualifiers prevent the creation of the wrapper, the Wrapping variant cannot be tested with qualifiers—a runtime error is raised when the wrapper is deemed necessary.

Results. Table 1 presents the results for the three variants. In each case, we compare the original gradual typing implementation (GT) with RCGT. For one millon iterations, the Fully Typed variant takes about 0.1 seconds in all scenarios. The No Wrapping variant takes about 11 seconds in GT, and 14 seconds in RCGT. This amounts to around a 100X slowdown compared to the Fully Typed variant, reflecting the cost of performing the function cast and associated subtyping tests. Note that the use of the $\uparrow$ qualifier implies a light 5% overhead, due to the tagging of the closure (Section 5.2). The Wrapping variant takes about 13 seconds in GT and about 17 seconds in RCGT. This means that wrap-

	Fully Typed	No Wrapping	Wrapping
GT	115	10800	12768
RCGT	120	13972	16721
RCGT with ↓	123	13761	-
RCGT with ↑	114	14412	-

Table 1. Execution time in milliseconds for one millon boundary crossings back into typed code. RCGT is about 30% slower than GT.

	No Wrapping	Wrapping
GT	2246	25458
RCGT	2648	26359
RCGT with ↓	2694	-
RCGT with ↑	2660	-

Table 2. Execution time in milliseconds for one million applications of the closure. Wrapper evaluation implies about a 10X slowdown.

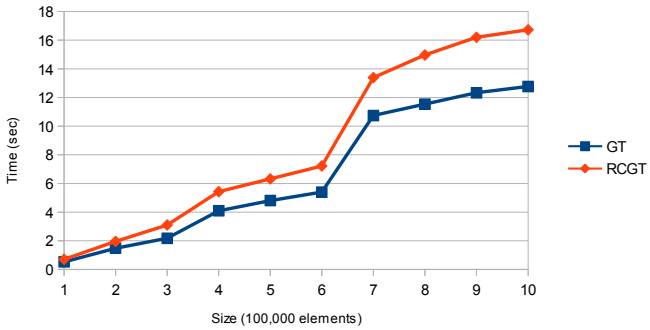

Figure 11. Running times for creating wrappers in GT and RCGT. RCGT is systematically about 30% slower.

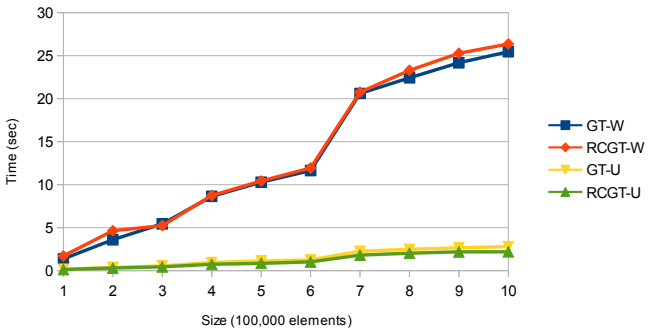

Figure 12. Running times for closure application when not wrapped (GT-U, RCGT-U) and wrapped (GT-W, RCGT-W). Wrappers always implies about a 10X slowdown.

per creation induces about a 20% slowdown, which is still negligible compared to the overhead of the casting logic.

RCGT turns out to be around 30% slower than GT, an observation that stands across all iteration counts (Figure 11). This overhead is due to the extra logic needed for function casts in RCGT compared to GT. We currently do not understand the reason of the discontinuity in the graph between 500k and 700k iterations, which is present in the data of all benchmarks.

6.1.2 The Cost of Applying Wrappers

To evalute the cost of applying wrappers, we benchmark two variants of a small program that repeatedly applies a block over a collection, after making it cross the typed/untyped boundary:

```
| block (Number Number → Number)tblock
 (TypedCollection<Number>)col |
block := [:(Integer)acc :(Integer)i|acc + i].
tblock := block.
col := self getCollection.
[col inject: 0 into: tblock] timeToRun
```

The third and fourth line create the block and perform the double boundary crossing. The fifth line obtains a collection filled with repetitions of the number 1 that is used for the fold operation. Its size hence determines the number of iterations in the benchmark loop. The last line performs the fold and measures the time.

The variants we consider are:

- **No Wrapping**: To establish a base line, this is the case where the block is not wrapped. It is obtained by changing the declared types of tblock, col and acc to be over Integer instead of Number.
- **Wrapping**: This is the variant in the above code snippet. The block is wrapped when assigning it to tblock, due to the mismatch of types.

Results. Table 2 reveals that, for 1 million iterations, application of a wrapped closure suffers roughly a 10X slowdown, both in GT as in RCGT. This relative overhead is preserved for all iteration counts (Figure 12). Also, the use of type qualifiers in the No Wrapping variant does not affect the performance of RCGT.

6.2 Macrobenchmark

To establish if the advantages of Relaxed Confined Gradual Typing scale up to a more real-world setting, we have performed a macrobenchmark using Spec [18]. Spec is the standard UI framework for Pharo Smalltalk, the language in which Gradualtalk is implemented. The focus of Spec is on reuse and composition of existing UI components, from basic widgets to complete user interfaces.

Setup The external interface of the standard Spec widgets essentially boils down to their configuration settings. In these widgets, a wide variety of settings are set by provid-

	Not Wrapped	Wrapped
GT	1113	1573
RCGT	1123	1608

Table 3. Execution time in milliseconds for the macrobenchmark. Wrapping induces a 45% slowdown.

ing a block closure, *e.g.* the action to perform when a button is pushed, or how to obtain the string representation of an object for a list view (known as its displayBlock). We have typed the external interface of these widgets to allow type errors, *e.g.* in the configuration blocks, to be caught at compile time. This instead of, *e.g.* when a user clicks on a button or a new item is added to the list and painted.

The macrobenchmark establishes the cost of using a wrapped displayBlock, compared to when it is not. This is done by showing a list of all 3620 classes present in the Gradualtalk image and programmatically scrolling down the entire list as fast as possible. Scrolling is performed by programmatically selecting each item in the list in succession. This scrolls the list down and whenever a new list item is painted the displayBlock is executed. The slower the execution of the block, the longer it will take to scroll through the entire list. The benchmark is run ten times and the average time is taken. Complete code is available online.

Results The macrobenchmark has four different variants: GT and RCGT, both with and without wrappers. The detailed results are in Table 3. When the block is wrapped, GT and RCGT take about 1.6 seconds, compared to 1.1 seconds when the block is not wrapped. In both cases RCGT is only 1-2% slower than GT. Overall, wrapping induces a 45% slowdown in scrolling speed, which is significant. Also, in the context of the experiment, the difference is well above the typical noticeable UI difference threshold of 0.2 seconds.

6.3 Summary

We have performed microbenchmarks that establish the cost of crossing the boundary back into statically typed code, and the cost of applying wrappers. We assessed both costs for GT and RCGT. Considering the boundary crossing, we found that RCGT is about 30% slower than GT, which is not significant compared to the 100X slowdown imposed by the casting logic in any case. More significant is that applying a wrapped closure is 10 times slower than an unwrapped one. The macrobenchmarks confirm that in a more real-life scenario the overhead of wrapped closure remains significant (45% for displaying a list widget). We conclude that avoiding unwanted wrapper creation through the use of $\uparrow$ and $\downarrow$ type qualifiers can have valuable performance benefits.

7. Related Work

If a programming language is to combine typed and untyped code safely, some form of casting is unavoidable. Prior work in this area has focused on decreasing the overhead that casts introduce. The Typed Racket language [25] combines static and dynamic code at a coarser granularity than the gradual typing approach. A Typed Racket module is either fully typed or fully untyped. This approach emphasizes program designs with fewer interaction points between static and dynamic code. The Confined Gradual Typing technique restricts the dynamic flow of some typed values into untyped code, regardless of how coarse or fine grained the borders between the two. As such, we believe that this technique is perfectly compatible with module-level approaches, and can be used to extend their expressive power.

The Thorn language [4, 28] decreases the performance overhead of casts by fine-tuning their granularity and the way they are checked. They introduce *like type* annotations on method parameters and return types. Method bodies are statically typed against the like-typed parameters, but not against their return type: Conversely, methods calls are statically checked only against the method return type. The remaining checks are performed dynamically. Like types are purely nominal and do not directly support higher-order casts, which must be encoded using objects.

Rastogi et al. [17], use local type inference to significantly reduce the amount of untyped code in a program and decrease the number of casts needed at runtime. This approach is automatic and effective. Confined Gradual Typing can extend inference-based approaches with more explicit programmer control over which values may ultimately be cast. Conversely, CGT could be extended with a form of qualifier inference in order to ease the task of annotating code with the strongest guarantees possible.

In addition to runtime cost, researchers have investigated techniques for reducing the space cost of casts. Herman et al. [12] observed that higher-order casts can accumulate at runtime and thereby compromise the space consumption of gradually typed programs that appear to be tail-recursive in the source language. They propose to use coercions to compress chains of casts dynamically. Going a step further, Siek and Wadler develop threesomes as a data structure and algorithm to represent and normalize coercions [21]. These space concerns are orthogonal to the value-flow concerns addressed by Confined Gradual Typing.

Recently, Swamy et al. [23] developed an alternative design for embedding gradual typing securely in JavaScript. All values are tagged with their runtime types. Performance and reliability issues are avoided by eagerly forbidding higher-order casts that would require wrappers. The approach is effective for first-order mutable objects, ensuring that there are no lazy cast errors in static code, but is admittedly too restrictive for higher-order patterns. CGT may be an interesting approach to recover some flexibility.

Finally, notions of blame and blame-tracking have been studied intently as a means to track down the source of dynamic errors in the face of higher-order casts [9, 15, 22, 26,

27]. These techniques are complementary to the Confined Gradual Typing approach, which gives the programer explicit control over which values or value-flows could become subject to implicitly-introduced higher-order casts.

8. Conclusion

Gradual typing appeals to programmers because it seamlessly and automatically combines typed and untyped code, while rejecting obvious type inconsistencies. This convenience, however, has its costs. Type casts smooth the boundaries between the typed and untyped worlds, but in higher-order languages these casts move about as a program runs, making it hard to predict which values will be wrapped and why. Confined Gradual Typing introduces type qualifiers to help programmers control which values can flow through the untyped world and be wrapped with casts in the process. Confined Gradual Typing can increase the predictability, reliability, and performance of gradually-typed programs.

Future work includes further developing the practical experience of Confined Gradual Typing to study and refine the typical usage scenarios through larger case studies. On the language design side, it is also interesting to consider alternatives like reverting the defaults (using qualifiers to *allow* boundary crossing), providing both the strict and relaxed variants in the same language, and generalizing the static/dynamic distinction to multiple parties like in TS* [23].

Acknowledgments

We thank the OOPSLA reviewers and the program committee for their insightful comments and suggestions.

References

[1] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, Aug. 2013. Available online.

[2] E. Allende, J. Fabry, and É. Tanter. Cast insertion strategies for gradually-typed objects. In *Proceedings of the 9th ACM Dynamic Languages Symposium (DLS 2013)*, pages 27–36, Indianapolis, IN, USA, Oct. 2013. ACM Press. ACM SIGPLAN Notices, 49(2).

[3] E. Allende, J. Fabry, R. Garcia, and É. Tanter. Confined gradual typing – extended version with detailed proofs. Technical Report TR/DCC-2014-3, University of Chile, May 2014.

[4] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2009)*, pages 117–136, Orlando, Florida, USA, Oct. 2009. ACM Press.

[5] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, pages 1–6, 2004.

[6] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, pages 215–230, Washington, D.C., USA, Oct. 1993. ACM Press. ACM SIGPLAN Notices, 28(10).

[7] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 278–292, Toronto, Ontario, Canada, 1991.

[8] G. Castagna, editor. *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, York, UK, 2009. Springer-Verlag.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, Pittsburgh, PA, USA, 2002. ACM Press.

[10] M. Furr. *Combining Static and Dynamic Typing in Ruby*. PhD thesis, University of Maryland, 2009.

[11] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, Nov. 2000.

[12] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher-Order and Sympolic Computation*, 23(2): 167–189, June 2010.

[13] L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*, pages 609–624, Portland, Oregon, USA, Oct. 2011. ACM Press.

[14] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2): Article n.6, Jan. 2010.

[15] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, pages 3–10, Nice, France, Jan. 2007. ACM Press.

[16] POPL 2010. *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2010)*, Madrid, Spain, Jan. 2010. ACM Press.

[17] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2012)*, pages 481–494, Philadelphia, USA, Jan. 2012. ACM Press.

[18] B. V. Ryseghem, S. Ducasse, and J. Fabry. Seamless composition and reuse of customizable user interfaces with Spec. *Science of Computer Programming*, 2014. To appear.

[19] J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

[20] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in Lecture Notes in Computer Science, pages 2–27, Berlin, Germany, July 2007. Springer-Verlag.

[21] J. Siek and P. Wadler. Threesomes, with and without blame. In POPL 2010 [16], pages 365–376.

[22] J. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In Castagna [8], pages 17–31.

[23] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in JavaScript. In *Proceedings of the 41st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2014)*, pages 425–437, San Diego, CA, USA, Jan. 2014. ACM Press.

[24] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*, pages 793–810, Tucson, AZ, USA, Oct. 2012. ACM Press.

[25] S. Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs*. PhD thesis, Northeastern University, Jan. 2010.

[26] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *Proceedings of the ACM Dynamic Languages Symposium (DLS 2006)*, pages 964–974, Portland, Oregon, USA, Oct. 2006. ACM Press.

[27] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In Castagna [8], pages 1–16.

[28] T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In POPL 2010 [16], pages 377–388.

A. SCGT: Correctness of Qualifiers

This appendix and the following contains abridged versions of the proofs. Detailed proofs are available in the companion technical report [3].

Taint-tracking internal language definition:

- syntax: Figure 13
- static semantics: Figure 14
- dynamic semantics: Figure 15

Note that the semantics (and the proofs) use the following notations to express predicates on marked values:

1. v°: Has never passed through Dyn (*i.e.* a value that underneath is marked with $\circ$)
2. $v^\bullet$: Has passed through Dyn (*i.e.* a value that underneath is marked with $\bullet$)
3. $\widehat{v}$: Can never pass through Dyn (*i.e.* a value that underneath is marked with $\widehat{\ }$)
4. v: can be any of the above

Lemma 1. *If $T_1 \rightsquigarrow \downarrow T_2$, then $T_1 <: \downarrow T_2$.*

Proof. There is only one rule in the direct consistency relationship that takes $\downarrow T_2$ in the right side(DC-down). And that rule requires that $\exists T_1', T_1 = \downarrow T_1'$ and $T_1' <: T_2$ to be able to use it. Using (SS-down), $T_1 <: \downarrow T_2$ □

Lemma 2. *(Inversion, taint tracking)*

e	::=	$n \mid \lambda x : T.e \mid \langle T \Leftarrow T \rangle\ e$	Expressions
	$\mid$	$e\ e \mid e + e$	
t	::=	$\circ \mid \bullet$	Taint
m	::=	$t \mid \widehat{t}$	Mark
s^m	::=	$n^m \mid \lambda^m x : T.e$	Marked primitive value
b	::=	$n^m \mid f$	Base values
f	::=	$\lambda^m x : T.e \mid \langle F \Leftarrow F' \rangle\ f$	Function values ($F' \not<: F$)
v	::=	$b \mid \langle \text{Dyn} \Leftarrow T \rangle\ b$	Values ($T \neq \text{Dyn}$)
P	::=	$B \mid \text{Dyn}$	Primitive Type
F	::=	$T \to T \mid \downarrow F \mid \uparrow F$	Function type
T	::=	$P \mid T \to T \mid \uparrow T \mid \downarrow T$	Type
E	::=	$\square\ e \mid v\ \square \mid \square + e$	Evaluation Frames
	$\mid$	$v + \square \mid \langle T \Leftarrow T \rangle\ \square$	

Figure 13. SCGT taint-tracking: Syntax.

1. *if* $\emptyset \vdash n^\circ : T$, *then* $T = \downarrow Int$
2. *if* $\emptyset \vdash n^{\widehat{\circ}} : T$, *then* $T = \updownarrow Int$
3. *if* $\emptyset \vdash n^\bullet : T$, *then* $T = Int$
4. *if* $\emptyset \vdash n^{\widehat{\bullet}} : T$, *then* $T = \uparrow Int$
5. *if* $\emptyset \vdash \lambda^\circ x : T'.e : T$, *then* $T = \downarrow(T_1 \to T_2)$
6. *if* $\emptyset \vdash \lambda^{\widehat{\circ}} x : T'.e : T$, *then* $T = \updownarrow(T_1 \to T_2)$
7. *if* $\emptyset \vdash \lambda^\bullet x : T'.e : T$, *then* $T = T_1 \to T_2$
8. *if* $\emptyset \vdash \lambda^{\widehat{\bullet}} x : T'.e : T$, *then* $T = \uparrow(T_1 \to T_2)$
9. *if* $\emptyset \vdash \langle F \Leftarrow F' \rangle\ f : T$, *then* $T = T_1 \to T_2$ *or* $T = \uparrow(T_1 \to T_2)$
10. *if* $\emptyset \vdash \langle Dyn \Leftarrow T \rangle b : T$, *then* $T = Dyn$

Proof. Direct from type rules, except Case v=$\langle F \Leftarrow F' \rangle f$.

Case v=$\langle F \Leftarrow F' \rangle f$

1. $F' \rightsquigarrow F$, $F' \not<: F$, by Well-typed value
2. $\emptyset \vdash \langle F \Leftarrow F' \rangle f : F$, by (TTT-cast)
3. $\emptyset \vdash \langle F \Leftarrow F' \rangle f : F$, $|F| = T \to T'$ by definition of F
4. $F \not<: \downarrow(T \to T')$, by contradiction between assumptions and Lemma 1
5. $\emptyset \vdash \langle F \Leftarrow F' \rangle f : F$, $F = T \to T'$ or $F = \uparrow(T \to T')$

□

Lemma 3. *(Canonical forms, taint tracking)*

1. $\emptyset \vdash v : T$ *and* $T <: \downarrow Int$, *then* $\exists n$, $v = n^\circ$
2. $\emptyset \vdash v : T$ *and* $T <: Int$, *then* $\exists n$, $v = n^\circ$ *or* $v = n^\bullet$
3. $\emptyset \vdash v : T$ *and* $T <: \uparrow Int$, *then* $\exists n$, $v = n$
4. $\emptyset \vdash v : T$ *and* $T <: \downarrow(T_1 \to T_2)$, *then* $\exists T', x, e$, $v = \lambda^\circ x : T'.e$ *and* $T <: T'$
5. $\emptyset \vdash v : T$ *and* $T <: T_1 \to T_2$, *then* $v = \lambda^\circ x : T'.e$ *or* $v = \lambda^\bullet x : T'.e$ *or* $v = \langle T_1 \to T_2 \Leftarrow F' \rangle f$
6. $\emptyset \vdash v : T$ *and* $T <: \uparrow(T_1 \to T_2)$, *then* $v = f$
7. $\emptyset \vdash v : Dyn$, *then* $v = \langle Dyn \Leftarrow T \rangle\ b^\circ$ *or* $v = \langle Dyn \Leftarrow T \rangle\ b^\bullet$

Proof. Direct from Lemma 2, except case $T = \text{Dyn}$

$$\text{(TTT-num1)}\frac{}{\Gamma \vdash n^\circ : \downarrow\text{Int}} \qquad \text{(TTT-num2)}\frac{}{\Gamma \vdash n^\bullet : \text{Int}} \qquad \text{(TTT-var)}\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(TTT-err)}\frac{}{\Gamma \vdash \text{CastError} : T}$$

$$\text{(TTT-abs1)}\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda^\circ x : T_1.e : \downarrow(T_1 \to T_2)} \qquad \text{(TTT-abs2)}\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda^\bullet x : T_1.e : T_1 \to T_2} \qquad \text{(TTT-}\uparrow\text{)}\frac{\Gamma \vdash s^t : T}{\Gamma \vdash s^{\widehat{t}} : \uparrow T}$$

$$\text{(TTT-app)}\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad |T_1| = T_{11} \to T_{12} \qquad T_2 <: T_{11}}{\Gamma \vdash e_1\ e_2 : T_{12}}$$

$$\text{(TTT-add)}\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 <: \uparrow\text{Int} \qquad T_2 <: \uparrow\text{Int}}{\Gamma \vdash e_1 + e_2 : \downarrow\text{Int}}$$

$$\text{(TTT-cast)}\frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2 \qquad T_2 \rightsquigarrow T_3}{\Gamma \vdash \langle T_3 \Leftarrow T_2 \rangle\ e : T_3}$$

Figure 14. SCGT taint-tracking: Typing.

$$\text{(TTE-congr)}\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \text{(TTE-app1)}\frac{T \neq \uparrow T'}{(\lambda x : T.e)\ v \longrightarrow e[v/x]} \qquad \text{(TTE-app2)}\frac{T = \uparrow T'}{(\lambda x : T.e)\ v \longrightarrow e[\widehat{v}/x]}$$

$$\text{(TTE-add)}\frac{n_3 = n_1 + n_2}{n_1 + n_2 \longrightarrow n_3^\circ} \qquad \text{(TTE-merge1)}\frac{T_1 \rightsquigarrow T_2 \qquad T \neq \text{Dyn} \qquad v \neq \widehat{v'}}{\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle v^\circ \longrightarrow \langle T_2 \Leftarrow |T_1|\rangle v^\bullet}$$

$$\text{(TTE-merge2)}\frac{T_1 \rightsquigarrow T_2 \qquad T \neq \text{Dyn} \qquad v \neq \widehat{v'}}{\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle v^\bullet \longrightarrow \langle T_2 \Leftarrow T_1\rangle v^\bullet} \qquad \text{(TTE-remove1)}\frac{T_1 <: T_2 \qquad T_2 \neq \uparrow T'}{\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow v}$$

$$\text{(TTE-remove2)}\frac{T_1 <: T_2 \qquad T_2 = \uparrow T'}{\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow \widehat{v}}$$

$$\text{(TTE-fcastinv)}\frac{|F| = T_1 \to T_2 \qquad |F'| = T_1' \to T_2' \qquad F \rightsquigarrow F' \qquad F \not<: F'}{(\langle F' \Leftarrow F\rangle\ f)\ v \longrightarrow \langle T_2' \Leftarrow T_2\rangle(f\ (\langle T_1 \Leftarrow T_1'\rangle\ v))}$$

$$\text{(TTE-err)}\frac{}{E[\text{CastError}] \longrightarrow \text{CastError}} \qquad \text{(TTE-merge-err)}\frac{T_1 \not\rightsquigarrow T_3}{\langle T_3 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle v \longrightarrow \text{CastError}}$$

Figure 15. SCGT taint-tracking: Evaluation.

Case $T = \textbf{\textit{Dyn}}$

1. $v = \langle \text{Dyn} \Leftarrow T\rangle b$, by Lemma 2-10
2. $T \rightsquigarrow \text{Dyn}$, by (TTT-cast)
3. $T \neq \uparrow T'$
4. $b \neq \widehat{v}$, by Lemma 3-1,2,4,5

□

Lemma 4. *If* $\emptyset \vdash v : T$ *and* $T <: \uparrow T$*, then* $\emptyset \vdash \widehat{v} : \uparrow T$

Proof. By case analysis of values:

Case $T = \textbf{\textit{Int}}$

1. $v = n$, by Lemma 3-3
2. $\emptyset \vdash \widehat{n} : \uparrow\text{Int}$, by (TTT-$\uparrow$)

Case $T = T_1 \to T_2$

1. $v = f$, by Lemma 3-6
2. $v = \lambda x : T_1.e$ or $v = \langle \uparrow T \Leftarrow T'\rangle\ f$
3. $v = \lambda x : T_1.e$:
   (a) $x : T_1 \vdash e : T_2$, well typed value
   (b) $\emptyset \vdash \widehat{\lambda} x : T_1.e : \uparrow T$, by (TTT-$\uparrow$)
4. $v = \langle \uparrow T \Leftarrow T'\rangle\ f$:
   (a) $\emptyset \vdash f : T'$, well typed value
   (b) $\langle \uparrow T \widehat{\Leftarrow} T'\rangle\ f : \uparrow T$, by (TTT-cast)

□

Theorem 8. *(Progress, taint tracking) If* $\emptyset \vdash e : T$*, then* e *is a value or* $e =$ *CastError or* $\exists e'$*,* $e \longrightarrow e'$*.*

Proof. By induction on the type rules.

Case (TTT-num1), (TTT-num2), (TTT-abs1), (TTT-abs2), (TTT-↑), (TTT-err), (TTT-var)

1. e is either a value or CastError, or the case is impossible

Case (TTT-app)

1. By assumption:
   (a) $\emptyset \vdash e_1\ e_2 : T_3$
   (b) $\emptyset \vdash e_1 : T_1, \emptyset \vdash e_2 : T_2, |T_1| = T_{11} \to T_{12}, T_2 <: T_{11}$
2. If e_1 is not a value, use rule (TTE-congr) with $E = \square\ e_2$ to progress.
3. If e_1 is CastError, use rule (TTE-err) with $E = \square\ e_2$ to progress.
4. If e_1 is a value and e_2 is not a value, use rule (TTE-congr) with $E = e_1\ \square$ to progress.
5. If e_1 is a value and e_2 is CastError, use rule (TTE-err) with $E = e_1\ \square$ to progress.
6. If e_1 and e_2 are both values, $e_1 = \lambda x : T'.e$ or $e_1 = \langle F \Leftarrow F' \rangle\ f$ by Lemma 3-6. Use (TTE-app1), or (TTE-app2) for the former or (TTE-fcastinv) for the latter to progress.

Case (TTT-add)

1. By assumption:
   (a) $\emptyset \vdash e_1 + e_2 : \downarrow\text{Int}$
   (b) $\emptyset \vdash e_1 : T_1, \emptyset \vdash e_2 : T_2, T_1 <: \uparrow\text{Int}, T_2 <: \uparrow\text{Int}$
2. If e_1 is not a value, use rule (TTE-congr) with $E = \square + e_2$ to progress.
3. If e_1 is CastError, use rule (TTE-err) with $E = \square + e_2$ to progress.
4. If e_1 is a value and e_2 is not a value, use rule (TTE-congr) with $E = e_1 + \square$ to progress.
5. If e_1 is a value and e_2 is CastError, use rule (TTE-err) with $E = e_1 + \square$ to progress.
6. If e_1 and e_2 are both values, then $e_1 = n_1$ and $e_2 = n_2$ by Lemma 3-3. Use (TTE-add) to progress.

Case (TTT-cast)

1. By assumption:
   (a) $\emptyset \vdash \langle T_2 \Leftarrow T_1 \rangle e : T_2$
   (b) $\emptyset \vdash e : T_1', T_1' <: T_1, T_1 \rightsquigarrow T_2$
2. If e is not a value, use rule (TTE-congr) with $E = \langle T_2 \Leftarrow T_1 \rangle\ \square$ to progress.
3. If e is CastError, use rule (TTE-err) with $E = \langle T_2 \Leftarrow T_1 \rangle\ \square$ to progress.
4. If e is a value and $T_1 = \text{Dyn}$, then $e_1 = \langle \text{Dyn} \Leftarrow T' \rangle\ v^\circ$ or $e_1 = \langle \text{Dyn} \Leftarrow T' \rangle\ v^\bullet$ by Lemma 3-7. Use rule (TTE-merge1), (TTE-merge2) or (TTE-merge-err) to progress.
5. If e is a value and $T_1 <: T_2$, use rule (TTE-remove1) or (TTE-remove2) to progress.
6. If e is a value and $T_1 \rightsquigarrow T_2$ and $T_1 \not<: T_2$, then $\exists F_1, F_2$ $T_1 = F_1$ and $T_2 = F_2$. Then $\langle T_2 \Leftarrow T_1 \rangle e$ is a value

□

Lemma 5. *(Substitution) If* $\Gamma, x : T \vdash e : T'$ *and* $\emptyset \vdash v : T''$, $T'' <: T$, *then* $\Gamma \vdash e[v/x] : T'''$ *and* $T''' <: T'$.

Proof. By induction on the type rules

Case (TTT-num1)

1. By assumption:
   (a) $\Gamma, x : T \vdash n^\circ : \downarrow\text{Int}$
2. $n^\circ[v/x] = n^\circ$ by Substitution definition
3. $\Gamma, x : T \vdash n^\circ[v/x] : \downarrow\text{Int}$, replacing (2) in (1-a)
4. $\Gamma \vdash n^\circ[v/x] : \downarrow\text{Int}$, by environment reduction.

Case (TTT-num2)

1. By assumption:
   (a) $\Gamma, x : T \vdash n^\bullet : \text{Int}$
2. $n^\bullet[v/x] = n^\bullet$ by Substitution definition
3. $\Gamma, x : T \vdash n^\bullet[v/x] : \text{Int}$, replacing (2) in (1-a)
4. $\Gamma \vdash n^\bullet[v/x] : \text{Int}$, by environment reduction.

Case (TTT-var), $y = x$

1. By assumption:
   (a) $\Gamma, x : T \vdash x : T, \Gamma(x) = T$
   (b) $\emptyset \vdash v : T', T' <: T$
2. $x[v/x] = v$, by Substitution definition
3. $\Gamma \vdash x[v/x] : T', T' <: T$ replacing (2) in (1-b)

Case (TTT-var), $y \neq x$

1. By assumption:
   (a) $\Gamma, x : T \vdash y : T', \Gamma(y) = T'$
2. $y[v/x] = y$, by Substitution definition
3. $\Gamma \vdash y[v/x] : T'$, replacing (2) in (1-a) and environment reduction.

Case (TTT-err)

1. By assumption:
   (a) $\Gamma, x : T \vdash \text{CastError} : T'$
2. $\Gamma \vdash \text{CastError} : T'$, by using (TTT-err)
3. $\Gamma \vdash \text{CastError}[v/x] : T'$, by Substitution definition

Case (TTT-abs1)

1. By assumption:
   (a) $\Gamma, x : T \vdash \lambda^\circ y : T'.e : \downarrow(T' \to T'')$
   (b) $\Gamma, x : T, y : T' \vdash e : T''$
2. Without loss of generality, $y \neq x$
3. $\Gamma, y : T' \vdash e[v/x] : T''$, by induction of (1-b)
4. $\Gamma \vdash \lambda^\circ y : T'.(e[v/x]) : \downarrow(T' \to T'')$, by (TTT-abs1)
5. $\Gamma \vdash (\lambda^\circ y : T'.e)[v/x] : \downarrow(T' \to T'')$, by Substitution definition

Case (TTT-abs2)

1. By assumption:
   (a) $\Gamma, x : T \vdash \lambda^\bullet y : T'.e : T' \to T''$
   (b) $\Gamma, x : T, y : T' \vdash e : T''$
2. Without loss of generality, $y \neq x$
3. $\Gamma, y : T' \vdash e[v/x] : T''$, by induction of (1-b)
4. $\Gamma \vdash \lambda^\bullet y : T'.(e[v/x]) : T' \to T''$, by (TTT-abs2)
5. $\Gamma \vdash (\lambda^\bullet y : T'.e)[v/x] : T' \to T''$, by Substitution definition

Case (TTT-↑)

1. By assumption:
   (a) $\Gamma, x : T \vdash s^{\widehat{t}} : \uparrow T'$
   (b) $\Gamma, x : T \vdash s^{t} : T'$
2. $s^{\widehat{t}} = s^{\widehat{t}}[v/x]$, by Substitution definition
3. $\Gamma \vdash s^{\widehat{t}}[v/x] : T'$, by (TTT-up)

Case (TTT-app)

1. By assumption:
   (a) $\Gamma, x : T \vdash e_1\ e_2 : T_{11}$
   (b) $\Gamma, x : T \vdash e_1 : T_1$
   (c) $\Gamma, x : T \vdash e_2 : T_2$
   (d) $|T_1| = T_{11} \to T_{12}, T_2 <: T_{11}$
2. $\Gamma \vdash e_1[v/x] : T_1$, by induction of (1-b)
3. $\Gamma \vdash e_2[v/x] : T_2$, by induction of (1-c)
4. $\Gamma \vdash e_1[v/x]\ e_2[v/x] : T_{12}$, by (TTT-app)
5. $\Gamma \vdash (e_1\ e_2)[v/x] : T_{12}$, by Substitution definition

Case (TTT-add)

1. By assumption:
   (a) $\Gamma, x : T \vdash e_1 + e_2 : \downarrow\text{Int}$
   (b) $\Gamma, x : T \vdash e_1 : T$
   (c) $\Gamma, x : T \vdash e_2 : T'$
   (d) $T <: \uparrow\text{Int}, T' <: \uparrow\text{Int}$
2. $\Gamma \vdash e_1[v/x] : T$, by induction of (1-b)
3. $\Gamma \vdash e_2[v/x] : T'$, by induction of (1-c)
4. $\Gamma \vdash e_1[v/x] + e_2[v/x] : \downarrow\text{Int}$, by (TTT-add)
5. $\Gamma \vdash (e_1 + e_2)[v/x] : \downarrow\text{Int}$, by Substitution definition

Case (TTT-cast)

1. By assumption:
   (a) $\Gamma, x : T \vdash \langle T \Leftarrow T' \rangle\ e : T$
   (b) $\Gamma, x : T \vdash e : T''$
   (c) $T' \rightsquigarrow T, T'' <: T'$
2. $\Gamma \vdash e[v/x] : T''$, by induction of (1-b)
3. $\Gamma \vdash \langle T \Leftarrow T' \rangle\ (e[v/x]) : T$, by (TTT-cast)
4. $\Gamma \vdash (\langle T \Leftarrow T' \rangle\ e)[v/x] : T$, by Substitution definition

□

Theorem 9. *(Preservation, taint tracking) If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T'$ and $T' <: T$.*

Proof. By induction on the evaluation rules.

Case (TTE-congr)

Subcase $E = \square\ e$

1. By assumption:
   (a) $e_1\ e_2 \longrightarrow e_1'\ e_2$
   (b) $\emptyset \vdash e_1\ e_2 : T_{12}, \emptyset \vdash e_1 : T_1, \emptyset \vdash e_2 : T_2, |T_1| = T_{11} \to T_{12}, T_2 <: T_{11}$
   (c) $e_1 \longrightarrow e_1'$
2. $\emptyset \vdash e_1' : T_1', T_1' <: T_1, |T_1'| = T_{11}' \to T_{12}'$, by induction on (1-c)
3. $|T_1'| <: |T_1|$, by using either (SS-reflex), (SS-gainup), (SS-losedown), (SS-down) or (SS-up)
4. $T_{11} <: T_{11}'$ and $T_{12}' <: T_{12}$, by using (SS-fun)
5. $T_2 <: T_{11}'$, by using (SS-trans)
6. $\emptyset \vdash e_1'\ e_2 : T_{12}', T_{12}' <: T_{12}$ by using (TTT-app) and (4)

Subcase $E = v\ \square$

1. By assumption:
   (a) $e_1\ e_2 \longrightarrow e_1\ e_2'$
   (b) $\emptyset \vdash e_1\ e_2 : T_{12}, \emptyset \vdash e_1 : T_1, \emptyset \vdash e_2 : T_2, |T_1| = T_{11} \to T_{12}, T_2 <: T_{11}$
   (c) $e_2 \longrightarrow e_2'$
2. $\emptyset \vdash e_2 : T_2', T_2' <: T_2$
3. $T_2' <: T_{11}'$, by using (SS-trans)
4. $\emptyset \vdash e_1\ e_2' : T_{12}$, by using (TTT-app)

Subcase $E = \square + e$

1. By assumption:
   (a) $e_1 + e_2 \longrightarrow e_1' + e_2$
   (b) $\emptyset \vdash e_1 + e_2 : T_3, \emptyset \vdash e_1 : T_1, \emptyset \vdash e_2 : T_2$
   (c) $e_1' \longrightarrow e_1''$
2. $T_1 <: \uparrow\text{Int}, T_2 <: \uparrow\text{Int}, T_3 <: \downarrow\text{Int}$ (From TTT-add)
3. $T_1' <: T_1$, by induction on (1-c)
4. $T_1' <: \uparrow\text{Int}$, by using (SS-trans)
5. $\emptyset \vdash e_1' + e_2 : T_3$, by using (TTT-add)

Subcase $E = v + \square$ Analogous to subcase $E = \square + e$

Subcase $E = \langle T \Leftarrow T \rangle\ \square$

1. By assumption:
   (a) $\langle T_2 \Leftarrow T_1 \rangle\ e \longrightarrow \langle T_2 \Leftarrow T_1 \rangle\ e'$
   (b) $\emptyset \vdash T_2 \Leftarrow T_1 \rangle\ e : T_2, \emptyset \vdash e : T_1', T_1' <: T_1$
   (c) $e \longrightarrow e'$
2. $\emptyset \vdash e' : T_1'', T_1'' <: T_1'$ by induction of (1-c)
3. $T_1'' <: T_1$, by (SS-trans)
4. $\emptyset \vdash \langle T_2 \Leftarrow T_1 \rangle\ e' : T_2$ (TTT-cast)

Case (TTE-app1)

1. By assumption:
   (a) $(\lambda x : T.e)\ v \longrightarrow e[v/x]$
   (b) $\emptyset \vdash (\lambda x : T.e)\ v : T', \emptyset \vdash v : T, T \neq \uparrow T''$
2. $\emptyset \vdash e[v/x] : T'', T'' <: T'$, by Substitution type preservation

Case (TTE-app2)

1. By assumption:
   (a) $(\lambda x : T.e)\ v \longrightarrow e[\widehat{v}/x]$
   (b) $\emptyset \vdash (\lambda x : T.e)\ v : T', \emptyset \vdash v : T, T = \uparrow T''$
2. $\emptyset \vdash \widehat{v} : T$, by Lemma 4
3. $\emptyset \vdash e[\widehat{v}/x] : T'', T'' <: T'$, by Substitution type preservation

Case (TTE-add)

1. By assumption:
   (a) $v_1 + v_2 \longrightarrow v_3^\circ$
   (b) $\emptyset \vdash v_1 + v_2 : T', v_3 = v_1 + v_2$
2. $T' = \downarrow$Int (From TTT-add)
3. $\emptyset \vdash v_3^\circ : \downarrow$Int (From TTT-num1)
4. $\emptyset \vdash v_3^\circ : T'$, replacing T'

Case (TTE-merge1)

1. By assumption:
   (a) $\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle\ v^\circ \longrightarrow \langle T_2 \Leftarrow |T_1|\rangle\ v^\bullet$
   (b) $T_1 \rightsquigarrow T_2$
2. $T_2 \neq \downarrow T_2'$, because Dyn $\not\rightsquigarrow \downarrow T_2'$
3. $\emptyset \vdash \langle \text{Dyn} \Leftarrow T_1\rangle\ v^\circ : \text{Dyn}$, reverse (TTT-cast) in (1-a)
4. $\emptyset \vdash v^\circ : T_1', T_1' <: T_1$ reverse (TTT-cast) in (2)
5. $\emptyset \vdash v^\bullet : |T_1'|$, by either (TTT-num2) or (TTT-abs2)
6. $|T_1'| <: |T_1|$, either by (SS-losedown) or (SS-down)
7. $|T_1| <: T_2$, by either (SS-reflex), (SS-losedown) or (SS-gainup)
8. $\emptyset \vdash \langle T_2 \Leftarrow |T_1|\rangle\ v^\bullet : T_2$ (TTT-cast)

Case (TTE-merge2)

1. By assumption:
   (a) $\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle\ v^\bullet \longrightarrow \langle T_2 \Leftarrow T_1\rangle\ v^\bullet$
   (b) $T_1 \rightsquigarrow T_2$
2. $\emptyset \vdash \langle \text{Dyn} \Leftarrow T_1\rangle\ v^\bullet : \text{Dyn}$, reverse (TTT-cast) in (1-a)
3. $\emptyset \vdash v^\bullet : T_1', T_1' <: T_1$ reverse (TTT-cast) in (2)
4. $\emptyset \vdash \langle T_2 \Leftarrow T_1\rangle\ v^\bullet : T_2$ (TTT-cast)

Case (TTE-remove1)

1. By assumption:
   (a) $\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow v$
   (b) $T_1 <: T_2, T_2 \neq \uparrow T_2'$
   (c) $\emptyset \vdash v : T_1$
2. $\emptyset \vdash v : T_1, T_1 <: T_2$

Case (TTE-remove2)

1. By assumption:
   (a) $\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow \widehat{v}$
   (b) $T_1 <: T_2, T_2 = \uparrow T_2'$
   (c) $\emptyset \vdash v : T_1$
2. $\emptyset \vdash \widehat{v} : \uparrow T_1'$, by Lemma 4
3. $\uparrow T_1' <: T_2$, (by SS-gainup)

Case (TTE-fcastinv)

1. By assumption:
   (a) $(\langle F' \Leftarrow F\rangle\ f)\ v \longrightarrow \langle T_2' \Leftarrow T_2\rangle(f\ (\langle T_1 \Leftarrow T_1'\rangle\ v))$
   (b) $|F| = T_1 \to T_2, |F'| = T_1' \to T_2', F \rightsquigarrow F', F \not<: F', T_1' \rightsquigarrow T_1, T_2 \rightsquigarrow T_2'$
   (c) $\emptyset \vdash (\langle F' \Leftarrow F\rangle\ f)\ v : T_2', \emptyset \vdash v : T$
2. $\emptyset \vdash \langle F' \Leftarrow F\rangle\ f : F'$, (TTT-cast)
3. $\emptyset \vdash f : F'', F'' <: F, |F''| = T_1'' \to T_2''$ inverse (TTT-cast)
4. $T_1 <: T_1'', T_2'' <: T_2$, by using (SS-fun)
5. $T <: T_1'$, inverse (TTT-app) of (1-c) using (2)
6. $\emptyset \vdash \langle T_1 \Leftarrow T_1'\rangle\ v : T_1$, (TTT-cast)
7. $\emptyset \vdash f\ \langle T_1 \Leftarrow T_1'\rangle\ v : T_2''$, (TTT-app)
8. $\emptyset \vdash \langle T_2' \Leftarrow T_2\rangle(f\ (\langle T_1 \Leftarrow T_1'\rangle\ v)) : T_2'$, (TTT-cast)

Case (TTE-err)

1. By assumption:
   (a) $E[\text{CastError}] \longrightarrow \text{CastError}$
   (b) $\emptyset \vdash E[\text{CastError}] : T$
2. $\emptyset \vdash \text{CastError} : T$, by using (TTT-err)

□

Theorem 10. *(↓ correctness) If $\emptyset \vdash v : \downarrow T$, then $v = v'^\circ$.*

Proof. Direct proof by using the Canonical form lemma. □

Theorem 11. *(↑ correctness) If $\emptyset \vdash \langle Dyn \Leftarrow T\rangle\ v : Dyn$, then $v \neq \widehat{v'}$.*

Proof. Direct proof by using the Canonical form lemma. □

B. RCGT- Correctness of Qualifiers

Lemma 6. *If $T_1 \rightsquigarrow \downarrow T_2$ and $T_1 \neq Dyn$, then $T_1 <: \downarrow T_2$.*

Proof. By induction on the Directed consistency rules that can take $T_1 \neq$ Dyn and $T_2' = \downarrow T_2$

Case (DC-losedown)

1. By assumption:
   (a) $T_1 = \downarrow T_1', T_1 \rightsquigarrow \downarrow T_2$
   (b) $T_1' \rightsquigarrow \downarrow T_2$
2. $T_1 <: T_1'$, by (SS-losedown)
3. $T_1' <: \downarrow T_2$, by induction on (1-b)
4. $T_1 <: \downarrow T_2$, by (SS-trans)

Case (DC-gainup)

1. By assumption:
   (a) $T_2 = \uparrow T_2', T_1 \rightsquigarrow \downarrow T_2$
   (b) $T_1 \rightsquigarrow \downarrow T_2'$
2. $T_1 <: \downarrow T_2'$, by induction on (1-b)by (SS-losedown)
3. $\downarrow T_2' <: \downarrow T_2$, by (SS-gainup)
4. $T_1 <: \downarrow T_2$, by (SS-trans)

Case (DC-sub) Direct from premise □

Lemma 7. *If* $\uparrow T_1 \rightsquigarrow T_2$ *and* $T_2 \neq$ *Dyn, then* $\uparrow T_1 <: T_2$

Proof. By induction on the Directed consistency rules that can take $T_2 \neq$ Dyn and $T_1' = \uparrow T_1$

Case (DC-losedown)

1. By assumption:
   (a) $T_1 = \downarrow T_1'$, $\uparrow T_1 \rightsquigarrow T_2$
   (b) $\uparrow T_1' \rightsquigarrow T_2$
2. $\uparrow T_1 <: \uparrow T_1'$, by (SS-losedown)
3. $\uparrow T_1' <: T_2$, by induction on (1-b)
4. $\uparrow T_1 <: T_2$, by (SS-trans)

Case (DC-gainup)

1. By assumption:
   (a) $T_2 = \uparrow T_2'$, $\uparrow T_1 \rightsquigarrow T_2$
   (b) $\uparrow T_1 \rightsquigarrow T_2'$
2. $\uparrow T_1 <: T_2'$, by induction on (1-b)
3. $T_2' <: T_2$, by (SS-gainup)
4. $\uparrow T_1 <: T_2$, by (SS-trans)

Case (DC-sub) Direct from premise □

We want to prove that a value typed as $\downarrow T$ has never been wrapped or a value typed as $\uparrow T$ will never be wrapped.

Theorem 12. *(No wrapping with qualifiers) If* $e = \langle T_2 \Leftarrow T_1 \rangle\ f$ *is a value and* $T_2 \neq$ *Dyn, then* $T_1 \neq \uparrow T_1'$ *and* $T_2 \neq \downarrow T_2'$

Proof. By contradiction

Case $T_1 = \uparrow T_1'$

1. By Lemma 7, $T_1 <: T_2$.
2. However, by definition of value, $T_1 \not<: T_2$. $\Rightarrow\Leftarrow$

Case $T_2 = \downarrow T_2'$

1. By Lemma 6, $T_1 <: T_2$.
2. However, by definition of value, $T_1 \not<: T_2$. $\Rightarrow\Leftarrow$

□

C. Microbenchmark Results

In these tables, time taken is reported in milliseconds and size means multiples of 100,000 collection elements.

Size	GT	RCGT	RCGT with ↓	RCGT with ↑
1	9.7	9.9	10.7	10.0
2	21.3	22.0	24.5	22.0
3	30.7	32.5	36.8	33.4
4	45.5	48.1	49.5	47.4
5	58.3	61.0	64.1	50.1
6	67.7	66.5	77.5	67.4
7	81.0	86.1	89.5	83.5
8	91.6	93.8	103.0	97.3
9	100.9	109.3	118.6	109.4
10	114.5	120.3	122.6	114.1

Table 4. Wrapper creation, Fully Typed benchmark.

Size	GT	RCGT	RCGT with ↓	RCGT with ↑
1	442.1	584.6	586.5	642.8
2	1173.8	1541.7	1527.3	1639.5
3	1618.5	2282.8	2243.8	2429.5
4	3319.9	4367.3	4299.1	4550.0
5	3749.1	4981.4	4898.0	5215.3
6	3972.8	5654.3	5455.2	5762.1
7	9323.9	11899.2	11783.0	12331.8
8	9995.8	13016.5	12902.1	13404.6
9	10540.8	13853.1	13745.9	14194.0
10	10800.9	13971.5	13761.1	14411.5

Table 5. Wrapper creation, No Wrapping benchmark.

	GT	RCGT
1	530.1	706.7
2	1479.5	1958.7
3	2175.0	3099.1
4	4094.7	5427.1
5	4803.9	6319.6
6	5394.6	7220.8
7	10742.6	13388.2
8	11535.6	14966.0
9	12329.2	16201.0
10	12767.7	16720.9

Table 6. Wrapper creation, Wrapping benchmark.

Size	GT	RCGT	RCGT with ↓	RCGT with ↑
1	120.2	119.3	119.5	120.2
2	298.5	279.8	301.7	310.0
3	423.2	467.1	477.9	499.2
4	745.8	799.1	832.1	834.5
5	848.4	1017.1	1036.8	1045.8
6	993.3	1207.0	1232.8	1231.6
7	1735.0	2065.4	2091.5	2115.0
8	1900.8	2328.7	2332.0	2321.0
9	2005.1	2514.4	2529.5	2511.1
10	2246.0	2648.0	2694.2	2660.3

Table 7. Closure evaluation, No Wrapping benchmark.

Size	GT	RCGT
1	1360.5	1713.9
2	3590.5	4634.8
3	5434.9	5236.3
4	8632.5	8739.1
5	10282.7	10421.1
6	11638.5	11936.7
7	20605.5	20751.2
8	22417.3	23291.6
9	24179.6	25280.2
10	25458.3	26359.4

Table 8. Closure evaluation, Wrapping benchmark.