UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPROVING THE EFFICIENCY AND RELIABILITY OF GRADUAL TYPING

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS MENCIÓN COMPUTACIÓN

ESTEBAN ARMANDO ALLENDE PRIETO

PROFESORES GUÍAS:
JOHAN FABRY
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
ALEXANDRE BERGEL
DAVID RÖTHLISBERGER
LAURENCE TRATT

Este trabajo ha sido parcialmente financiado por CONICYT

SANTIAGO DE CHILE
2015

Resumen

Gradual Typing permite a un programador aplicar tipos estáticos a ciertas partes de un programa, dejando el resto dinámicamente tipeado. Sin embargo, esto viene con un costo en el rendimiento. Una razón es que el runtime tiene que realizar siempre un casteo en el borde entre tipos estáticos y dinámicos. Otra razón es que el borde puede ser creado accidentalmente. Esto también trae un efecto lateral de reducir la fiabilidad del código estático, porque ahora el programador no puede garantizar que su código no arrojará errores de tipo en tiempo de ejecución.

En este trabajo de tesis, mejoramos el rendimiento y la fiabilidad de los programas gradualmente tipeados. Para esto, desarrollamos un lenguaje gradualmente tipeado, Gradualtalk, y luego presentamos dos novedosas ideas: hybrid strategy y Confined Gradual Typing.

La hybrid strategy es una nueva forma de insertar los casts al invocar métodos que combina dos estrategias existentes, permitiendo obtener el mejor rendimiento de ambas. Validamos esta afirmación con benchmarks.

Confined Gradual Typing refina gradual typing con anotaciones para prohibir explícitamente ciertos cruces de frontera entre el código estáticamente y dinámicamente tipeado. Nosotros desarrollamos formalmente dos variantes de CGT que capturan diferentes compromisos entre flexibilidad/garantías. Probamos que CGT es type sound y que las anotaciones ofrecen las garantías esperadas.

Abstract

Gradual typing allows a programmer to apply static typing to certain portions of a program, leaving the rest dynamically typed. However, this comes with a performance cost. One reason is that the runtime needs to perform a typecheck in the boundary between static and dynamic typing. Another reason is that the boundary can be created accidentally. This also have the side effect of reducing the reliability of static code, because now the programmer cannot ensure that his code will not throw type errors at runtime.

In this thesis work, we improve the performance and reliability of gradually typed programs. To adress this, we develop a gradually-typed language, Gradualtalk, and then we present two novel ideas: hybrid strategy and Confined Gradual Typing.

The hybrid strategy is a novel way to insert casts on method invocation that combines two existing strategies, allowing to get the best performance of each. We validate this claim with benchmarks.

Confined Gradual Typing refines gradual typing with annotations to explicitly prohibit certain boundary crossings between statically and dynamically typed code. We formally develop two variants of CGT that capture different flexibility/guarantee tradeoffs. We prove that CGT is type sound and the qualifiers provide the expected guarantees.

Agradecimientos

En primer lugar, deseo agradecer a mis profesores guías Johan Fabry y Éric Tanter por la enseñanza, el apoyo y la paciencia mientras realizaba mi doctorado. Muchas gracias por ayudarme a ampliar mi modo de visión al de investigador, cambiando mi forma de ver el mundo profesional.

Quiero agradecer a mi país, que a través de CONICYT financio mis estudios de doctorado. Además deseo agradecer al gobierno francés, que a través de su embajada en Chile, financio mi pasantía en Francia. También agradezco a todas las personas en INRIA con quienes trabaje mientras realizaba mi pasantía, en particular a Marcus Denker.

Igualmente, quiero darle las gracias a mis compañeros de posgrado que me han acompañado en esta travesía. Algunos de ellos son: Oscar Callau, Paul Ledger, Teresa Bracamonte, Daniel Moreno, Ismael Figueroa y Héctor Ferrada. Perdónenme a los que he olvidado. Además agradezco al personal académico y administrativo del DCC. En particular, a Angélica Aguirre y Sandra Gáez, que me han ayudado a destruir cada una de las rocas administrativas que me he encontrado en cada paso de mi viaje por el doctorado.

Finalmente, quiero agradecer a mi familia: a mi madre María Angélica Prieto, mi padre José Allende y hermano Sebastián Allende; quienes siempre me han apoyado y han estado conmigo. Muchas gracias por el apoyo tanto emocional como económico.

Table of contents

Chapter 1

Introduction

1.1 Introduction

While developing applications, software engineers are always looking for reliable methods of program verification (*e.g.* functions behave correctly with respect to some specifications) at several levels of modularity (*e.g.* expression, function, module). Among all the methods in the literature, such as runtime monitoring, Hoare logic [31], model checking [17], denotational semantics [56] and others, *type systems* [40] have arguably become the best known and the most used in practice while being a relatively lightweight formal method.

A type system is a tractable method for proving the absence of certain program behaviors [41].A type system classifies program expressions according to the kinds of values they evaluate to. These kinds of values are called types. There are two principal kinds of type systems that define how a given program is validated. These are static type systems and dynamic type systems. Each one has its own features, advantages and drawbacks.

1.1.1 Static Type Systems

A static type system can be regarded as calculating a static approximation of the runtime values of all the terms in a program. Languages such as C, Java, and Haskell use this kind of type system.

Statically typed languages enforce types at compile time. These languages are generally explicitly typed, *i.e.* types are part of their syntax (*e.g.* Java and C). Other languages such as ML and Haskell are implicitly typed: they include an inference process that assigns types to well-formed expressions, making type annotations optional.

Static type systems represent a highly valuable first line of defense against programming errors, because they can catch type errors early in the development cycle. Static type systems use the type information to verify the absence of some bad program behaviors (*e.g.* invocation of a method that is not implemented in the receiver object). Moreover, in a statically typed language, efficiency improvements can be obtained by eliminating many of the dynamic checks that would be needed to guarantee a type-safe execution in a dynamically typed language [41].

However, static type systems have a number of drawbacks as reported by Tratt [61]:

- They are not flexible and a change in the software requirements could require redesigning the software. Static type systems often prevent software evolution, because they require that the program as a whole always typechecks correctly. So, it is not possible to turn off selectively static type-checking.
- They are too conservative; meaning that, certain valid programs will not be accepted by the type checker.
- Relatively small increases in the expressiveness of static type systems cause a comparatively large increase in language complexity.

The above disadvantages can be solved, at least partially, by using a dynamic type system (see Section 1.1.2) or by deferring part of the checking to runtime. For example, several languages, such as Java, support explicit type coercions (called *casts*), whose effect is to allow a programmer to specify the type of the expression, and run a type check at runtime. So, `Point p = (Point) list.getFirst()` gets the first element of a list and casts it to `Point`, allowing to bypass the restriction of storing in p an object that must be proven to be of type Point at compile time. However, at runtime, a check is performed in order to verify that the returned object is actually of type Point. If not, a runtime exception is thrown.

1.1.2 Dynamic Type Systems

Dynamic typing, also referred to as "dynamic checking" [13], differs from static typing on how types are enforced. While a static type system enforces types at compile time, a dynamic type system performs type checks at runtime. Dynamically typed languages use tags, an annotation on the value about its type, at runtime to distinguish different kinds of structures, so they can guarantee safe execution [36].

There are many languages that are dynamically typed, *e.g.* Smalltalk, JavaScript, Python, and others. They are typically used in the industry to allow agile support for systems and fast adaptation to changing requirements [36]. Features like the ones described before make dynamically typed languages prominent in modern software development [36]. In general, dynamically typed languages allow code to be written that is more expressive than what would be possible with typical statically typed languages [61]. This is because a written code in a statically typed language requires that the programmer could also express the required types, and this restriction is lifted in a dynamically typed language.

Nevertheless, applying dynamic type systems have some drawbacks as reported by Tratt [61]:

- Software developed in languages that are dynamically typed are generally slower than its equivalent written in a statically typed language.
- Dynamic type systems cannot emit warnings or detect errors until the execution of the program. That means some errors that would easily be found at compile time, could eventually even reach the production stage of the software, increasing the cost for fixing them.
- Statically typed programs provide an enforced form of documentation. It is possible to informally annotate the expected types of a function in comments, but these neither represent any guarantee nor are they usually updated if the function specification changes. Some features of sophisticated Integrated Development Environments (IDEs), such as code completion, are enhanced by type information; such features cannot be provided easily (if at all) for dynamically typed languages.

Static and dynamic typing seem to be antagonists, but they actually are complementary. The following questions naturally arise: is it possible to integrate both? If so, how can that integration assist programmers in the software development process?

1.1.3 Integrating Static and Dynamic Type Systems

The main difference between statically typed and dynamically typed languages is the stage at which types are enforced. Actually we can see them as complementary: the advantages of one of them are the drawbacks of the other. Integrating both is the next logical step, and it has been pursued for some time. From the *dynamic type*, mentioned in Abadi *et al.* [1], to most modern partial typing techniques such as *like types* [63], researchers have been studying this particular symbiosis.

Listing 1.1: Gradual typing example

```
1  ModuleSystem class>> (Self) add: (Module)m
2    modules add: m.
3
4  Loader >> loadFromNetwork
5    |toLoad|
6    toLoad := Network getObject.
7    toLoad isModule ifTrue: [
8      ModuleSystem add: toLoad.
9    ]
10   ...
```

Gradual typing [50, 51] is one of those techniques. It combines the advantages of both static and dynamic type checking. Gradual typing allows a programmer to apply static typing to certain portions of a program, leaving the rest dynamically typed. This enables the programmer to select the best trade-off between the advantages and disadvantages between static typing and dynamic typing.

Listing 1.1 is an example of a gradually typed program written in Gradualtalk [1], a gradually typed Smalltalk, introduced in Chapter 3. It shows two methods: The first in Line 1 and Line 2, while the second method is between Line 4 and Line 10. Line 1 is the method declaration of the first method: we define the method add: that is a class method of ModuleSystem. It accepts one argument, m, of type Module and returns an object of type Self. Line 2 adds the argument that the method

[1]The webpage for Gradualtalk is http://www.pleiad.cl/gradualtalk

received to the collection stored in the class variable modules. Line 4 is the method declaration of the second method: we are defining the method loadFromNetwork that is an instance method of Loader. It accepts no arguments and returns a dynamically typed value. Line 5 declares one local variable named toLoad that is dynamically typed. Line 6 calls the class method getObject of the class Network and stores the result in toLoad. Line 6 checks that the object is a module calling the method isModule and if that is the case, executes Line 7. Line 7 calls the class method add: of ModuleSystem, implemented in Line 1 and Line 2, passing as parameter the value stored in the variable toLoad.

While all methods in the class `ModuleSystem` are statically typed, the methods in class `Loader` are dynamically typed. The only visible interaction between these worlds in this example is the call to the method `add` in `loadFromNetwork`. The places where the program flow passes from a dynamically typed world to a statically typed world, or vice versa, is the boundary of these worlds. At the boundary, there is a need to perform a type check only when passing from a dynamically typed world to a statically typed world. These dynamic type checks are implicitly introduced by the compiler/type checker. The compiler/type checker decides where the type check goes in the code. In the example before, there is an implicit type check in line 8 that the variable `toLoad` is actually a subtype of class `Module` when calling the method `add` in class `ModuleSystem class`.

1.2 Problem statement

Static and dynamic typing have different strengths, making them suited for different tasks. Static typing provides for efficient program execution, compile-time error detection, and better documentation, whereas dynamic typing enables rapid development and fast adaptation to changing requirements [61]. Gradual typing [50, 51] allows a programmer to apply static typing to certain portions of a program, leaving the rest dynamically typed. This enables us to select the best trade-off between the advantages and disadvantages of static typing and dynamic typing.

However, the advantages of efficient program execution due to static typing do not immediately carry over to a gradual type system. This is in part because of the existence of the boundary between statically typed and dynamically typed code.

The existence of this boundary poses two problems, which are specific to gradual typing. The first problem is that the runtime needs to perform a type check in the boundary; otherwise, it would be type unsafe. This check implies a performance penalty in the program. The second problem is that the boundary depends on the execution flow of the program, instead of on the lexical structure of the source code. For example, when a dynamically typed piece of code calls a function whose code has been fully typed, a type check must be performed to trap errors early before executing the function. However, when a statically typed piece of code calls the same function, there is no need to do a type check.

The existence of the boundary makes that type checks at runtime are done for a different reason in dynamically typed programs than in gradually typed programs. In a dynamically typed application, if the program would never raise an error, a perfect optimizer could discard all runtime checks. However, in the case of a gradually typed application, this is not possible because an object having an incorrect type must not pass the boundary to the statically typed world, even if the program would run fine otherwise. These differences in semantics makes discarding type checks as done by some dynamically typed languages (*e.g.* Self), difficult on gradually typed languages.

The existence of the boundary and the associated performance penalty discussed above, makes the decision to use gradual typing over dynamic typing not so easy when performance is critical. Furthermore, to the best of our knowledge, no published research has focused in reducing the performance hit of gradually typed applications with higher order (function) casts, except indirectly by optimizing the cast themselves.

Sometimes, the boundary is created accidentally by the programmer. For example, he could have forgotten to type one instance variable. Because of that oversight, that instance variable will be assigned the type `Dyn`, a special type assigned to dynamically typed values. If the rest of the program is typed, then all read or write to that instance variable will need to cross the boundary between statically typed and dynamically typed code.

Even when the existence of this boundary is accidental, it still affects the performance as we mentioned above. But it also affects the reliability of static code. This is because the programmer can never be sure that his code is 100% statically typed nor that values managed by his code are not wrapped, even when his code is 100% annotated. When closures flow from or into the untyped world, they may be wrapped to check at runtime that their arguments and the return value have the correct type. However, evaluating these wrappers has a cost in performance. The programmer does not have the tools that could help him to prevent accidentally creating such wrappers.

1.3 Goals of this work

The main goal of this thesis work is to increase the performance and improve the reliability of gradually typed applications, using techniques transferable to any gradually typed language. As a result, these techniques cannot be dependent on the presence of certain optimizations, like Just-In-Time compilers. To achieve our main goal, there are three specific goals:

- Implement a gradually typed language. The main motivation for this goal is to have a gradually typed language where we can implement our proposed optimizations and validate them.
- Reduce the performance penalty suffered in the boundary between statically typed and dynamically typed sections when invoking methods.
- Improve the reliability and remove the existence of accidental boundaries by giving programmers the tools to allow or disallow accidental crossing between statically and dynamically typed code.

1.4 Methodology

We address the three specific goals of this thesis work as follows. The first goal is addressed with Gradualtalk. Gradualtalk is the gradually typed Smalltalk we developed. In this thesis, we present the design and implementation of Gradualtalk.

The second goal is addressed introducing a new cast insertion strategy: the hybrid strategy. In this thesis, we describe two insertion strategies for method invocation casts and propose the hybrid strategy, which is a merge of the other two strategies. We also validate our claim that the hybrid strategy provides the best performance when invoking methods in all situations, compared with the other two cast insertion strategies. Using the hybrid strategy, we can reduce the performance penalty suffered in the boundary between statically typed and dynamically typed sections when invoking methods. The speedup gained heavily depends on how the program is typed and the strategy that is compared. The speed up can be between 0% and 93% less time spent in total.

The third goal is addressed with Confined Gradual Typing. Confined Gradual Typing defines two new qualifiers for types, which are the tools the programmers have to control boundary crossing between statically and dynamically typed code. These improve the reliability and remove the existence of accidental boundaries, because they allow to control whether values are able to cross the boundary. One qualifier prohibits crossing the boundary in the past, while the other prohibits crossing the boundary in the future. We develop two variants of Confined Gradual Typing. In the first variant, the qualifiers allow to control whether the values can flow from or to the dynamically typed code. In the second variant, the qualifiers allow to control whether the values have never been wrapped or cannot be wrapped.

1.5 Contributions

In this work, we make three contributions: Gradualtalk, Cast Insertion Strategies and Confined Gradual Typing. Each of these contributions have resulted in the following publications:

- **Gradual Typing for Smalltalk** [2]: This journal paper describes Gradualtalk and its features, as well as a preliminary empirical validation of its design (not included in this thesis).
- **Cast Insertion Strategies for Gradually-Typed Objects** [4]: This conference paper describes the different cast insertion strategies and validates them.
- **Confined Gradual Typing** [3]: This conference paper presents Confined Gradual Typing (CGT) in two variants. The paper provides a formal descrip-

tion and proof for each variant. Finally, it presents an implementation for CGT and shows that there is no significant performance hit for using CGT.

1.6 Structure of this thesis

This thesis is organized as follows. In Chapter 2, we present the necessary background of gradual typing. In Chapter 3, we introduce Gradualtalk, a gradually typed language, and provide its design and implementation. In Chapter 4, we study two cast insertion strategies and introduce a new cast insertion strategy, validating it versus the other two. In Chapter 5, we presents Confined Gradual Typing, an extension to gradual typing that allows to control the flow between statically typed and dynamically typed code, and provide formal proof that it is type sound and actually provides the guarantees we expect. In Chapter 6, we give a summary of this work and provide perspectives for future work.

Chapter 2

Gradual Typing

Gradual typing is a type discipline where expressions are either statically typed or dynamically typed, allowing programmers to choose either type discipline in different sections of the program code. To make this integration possible, gradual typing introduces a special type: Dyn. The Dyn type represents statically-unknown types. Gradual typing also introduces and changes type and evaluation rules to specify the meaning of this unknown type.

In this chapter we introduce gradual typing, how it is implemented, extensions to gradual typing and related work.

In explanations in this chapter, the language used to describe has both objects and closures. For simplicity's sake, we treat objects as black boxes and there is a nominal subtype relationship that relates them. A type T can be either a nominal type A, the unknown type Dyn or a function type $T_1 \to T_2$.

2.1 Gradual Typing in a Nutshell[1]

Gradual typing [50, 51] allows for the smooth integration of static and dynamic typing by swapping the conservative pessimism of a static type system (*i.e. rejecting all programs that may go wrong*) for a healthy dose of optimism (*i.e. accepting all programs that may go right*).

[1]This section is based on Section 2.1 of the paper "Confined Gradual Typing" [3]

Suppose an untyped function id, and two variables s1 and s2 that are both statically typed as String. The programmer can reasonably expect s2 = id s1 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 type of id is the unknown type, denoted Dyn, and Dyn is *consistent* [50] with any type. Consistency and consistent subtyping will be described in more detail in Section 2.2.2.

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 s2 is of type String is not violated. Internally, the above program is rewritten to an intermediate representation in which *casts* (<Target type ⇐ Source type>) are inserted:

```
s2 = <String ⇐ Dyn> id (<Dyn ⇐ String> s1)
```

The first cast executed, from String to Dyn, serves no purpose in a language whose runtime is based on tagged values (*i.e.* values with annotations about its type). In a language with untagged values, it tags the value referred to by s1 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, casts to function types, are much more subtle. Consider a variation of the above example in which s1 is typed as String → String and s2 has type X. The intermediate representation with casts is now as follows:

```
s2 = <X ⇐ Dyn> id (<Dyn ⇐ String → String> s1)
```

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 obeys the type restrictions of 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 s1. 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:

```
s2 = λ x: A. <B ⇐ String>(v <String ⇐ A>x)
```

Note that if A=B=String, then the corresponding wrapper would always succeed trivially. This allows the wrapper to be avoided altogether, in which case s2 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.

2.2 Implementation

Until now, we have seen the programmer's view of gradual typing. In this section, we discuss how gradual typing is implemented. The implementation of gradual typing can be divided in two parts: the type checker and the cast insertion process. However, before we can discuss each of these parts, we need to introduce the concept of consistency and consistent subtyping.

The implementation explained in this section corresponds to the work developed by Siek and Taha [50, 51].

2.2.1 Consistency

Gradual typing needs a concept of type substitutability between static types and the Dyn type to allow the seamless interaction between them. A value typed as Dyn can be used in any part where any other type is expected. In the same way, any type can be used where a Dyn typed value is expected.

The naive solution would be to make Dyn a subtype of any type ($\mathsf{Dyn} <: T$) and all types a subtype of Dyn ($T <: \mathsf{Dyn}$). However, because the subtype relationship is transitive, this would allow for any type to be a subtype of another arbitrary type ($T_1 <: \mathsf{Dyn} <: T_2$). This would break the purpose of the type checker to disallow values to be used incorrectly when they are of a statically-incompatible types. The solution proposed by Siek and Taha [50] is to introduce a new relationship between types: consistency.

The consistency relationship ($\sim$) is the type equivalency relationship for gradual typing, relating two types that are "equivalent" in gradual typing. This relaxes type equality: if two types are consistent, one type can replace another in any context and viceversa. In gradual typing, any type is consistent with Dyn and viceversa.

Formally, the consistency relationship is defined using the following four rules:

1. $\mathsf{A} \sim \mathsf{A}$
2. $\mathsf{A} \sim \mathsf{Dyn}$
3. $\mathsf{Dyn} \sim \mathsf{A}$
4. $T_{11} \rightarrow T_{12} \sim T_{21} \rightarrow T_{22}$, if $T_{11} \sim T_{21}$ and $T_{12} \sim T_{22}$

2.2.2 Consistent subtyping

The consistency relationship is sufficient for a functional language. However, in a type system with subtyping, like the ones used in object oriented languages, it is not sufficient. This is because the subtyping relationship is the relationship that relates what types can replace another type. The solution proposed by Siek and Taha [51], when there is subtyping, is to introduce another relationship: consistent subtyping.

The consistent subtype relationship ($\lesssim$) relates a type T_1 with a type T_2. If T_1 is a consistent subtype of T_2, T_1 can substitute T_2 in gradual typing with subtyping. This relationship is reflexive, but not transitive. This prevents the problem of the naive solution described in Section 2.2.1.

The consistent subtype relationship is defined using both consistency and subtyping. Informally, a type T_1 is a consistent subtype of T_2, iff T_1 is a subtype of T_2 using only once the consistency relationship instead of the equals operator. Formally, the consistent subtype relationship is defined using the following two rules:

1. $\mathsf{A} \lesssim \mathsf{B}$, if $\mathsf{A} <: \mathsf{B}$ or $\mathsf{A} \sim \mathsf{B}$.

2. $T_{11} \to T_{12} \lesssim T_{21} \to T_{22}$, if $T_{21} \lesssim T_{11}$ and $T_{12} \lesssim T_{22}$

2.2.3 Gradual typing type checker

The goal of the type checker is to reject obviously wrong programs (*e.g.* assigning a String to an Integer variable) at compile time. In the case of gradual typing, the type checker must be aware of the existence of the Dyn type and allow the interaction between statically typed variables and dynamically typed variables.

To make a normal type checker of either a functional language or object oriented language aware of the Dyn type, therefore making it a gradual type checker, two changes must be made. The first change is to use consistency instead of equality and consistent subtyping instead of subtyping in the typing rules.

The second change is to take care of rules that use information that would not be available if the type is Dyn. The only rule with this issue in functional languages is the application rule:

$$\text{(T-APP)}\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 \sim T_{11} \to T_{12} \qquad T_2 \lesssim T_{11}}{\Gamma \vdash e_1\ e_2 : T_{12}}$$

This type rule checks that the argument passed to the function is a subtype of its parameter type. If the type of the function T_1 is Dyn, then the parameter type T_{11} and the return type T_{12} could be any type and this rule would be non-deterministic. To solve this, the rule is broken up into two rules, one for each case:

$$\text{(T-APP1)}\frac{\Gamma \vdash e_1 : \text{Dyn} \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : \text{Dyn}}$$

$$\text{(T-APP2)}\frac{\Gamma \vdash e_1 : T_{11} \to T_{12} \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 \lesssim T_{11}}{\Gamma \vdash e_1\ e_2 : T_{12}}$$

In the case when the type of the function T_1 is Dyn, the dependent types of T_1 (*i.e.* T_{11} and T_{12}) are assumed to be Dyn too. In the case of the rule (T-APP1), the condition $T_2 \lesssim \text{Dyn}$ is redundant because it is always true, so it has been removed from the rule.

2.2.4 Cast Insertion process

The type checker has blind faith with regard to the interaction between statically typed values and dynamically typed values. However, at runtime we cannot have this blind faith, because that would make the type system unsound. One solution would be that the runtime checks everywhere if the value is of the expected type. However, this would unnecessarily and drastically reduce the performance. The solution proposed by Siek and Taha [51] is to insert casts in places where it is known that an interaction between the statically typed and dynamically typed code happens. This corresponds to two kinds of places in the code.

The first kind of place is when the typecheck rule is only true using consistency or consistent subtype, but becomes false if the check uses equality or subtyping, respectively, as a non-gradual type checker would. For example, when checking the rule (T-APP2), if $T_2 \lesssim T_{11}$ but $T_2 \not<: T_{11}$. In this case, a cast is inserted around the value that fails this check from its type to the target type. In the example, the expression $e_1\ e_2$ would be translated to $e_1\ (\langle T_{11} \Leftarrow T_2\rangle\ e_2)$

The second kind of place is when the type checker needs to be too permissive if a value has type Dyn. This is the case for the rule (T-APP1). Because the function has type Dyn, the runtime needs to check that e_1 is a function that can accept the argument that is passed. To carry out this check, the expression $e_1\ e_2$ is always translated to $(\langle T_2 \to \text{Dyn} \Leftarrow \text{Dyn}\rangle\ e_1)\ e_2$ when the rule (T-APP1) is used to typecheck that expression.

2.3 Extensions to Gradual Typing

The work of Siek and Taha [50, 51] is the foundation of gradual typing. However, there are still further work that extends gradual typing to support additional features. In this section, we discuss those additional works.

2.3.1 Generics

Ina *et al.* extend gradual typing to support generics [32]. Extending gradual typing to a type system with generics is straightforward. Similar to what we explained in Section 2.2.3, the rules with generics must use consistent subtyping instead of subtyping. However, integrating generics also requires one more decision: the default type when no type is defined for its generics. For statically typed languages, the default type is the root of the object type system (*e.g.*Object for Java).

For gradually typed languages, a better default type is the Dyn type. This is because dynamically typed code can obtain types without defined generics when creating a new object from a class constant (*e.g.*Collection new). If the default type were Object, it would produce typecheck errors in code that should run when the code is completely dynamically typed. For example, in a gradually typed Smalltalk, the code `(Collection new add: 3) first + 32` would raise an error, because the retrieved element from the created collection would be of type Object, and Object typed values do not support the addition operator.

2.3.2 First-class classes

Takikawa *et al.* developed a gradual type system for first-class classes in Racket [55]. They notice that the calculus of Siek and Taha does not deal with inheritance, and that a direct adaptation would be unsound. This is because an overriden method could change his return or parameter type from or to Dyn, which would not be correctly checked. Furthermore, the inheritance semantics that their work supports is rich in that it deals with accidental overridings. In order to address both requirements, their work adopts row polymorphism [45] instead of standard subtype polymorphism. To match these static typing features, on the dynamic checking side they propose opaque and sealed contracts. Opaque and sealed contracts protect methods or fields of classes from being accessed or overriden if they are not explicitly declared in the contract. A sealed contract can be removed from a class by an user, thus removing the protection if the user has the seal key. In contrast, an opaque contract protection cannot be removed after being applied. Consequently, mixins and other higher-order programming patterns with first-class classes can be checked soundly.

2.3.3 More than one kind of dynamically typed code

A language can use a type checker to provide more guarantees than preventing unhandled errors. Security is one of these guarantees. From the perspective of security protection, there are two kinds of dynamically typed code. One kind of dynamically typed code is one that is still under the scrutiny of the gradual typechecker. The other kind of dynamically typed code is code that has not been verified by the typechecker.

Swamy *et al.* [54] proposes a trichotomy between statically typed code, checked dynamically typed code and unchecked code. The type Dyn represents a value from checked dynamically typed code, while a new special type, Untyped, represents a value from unchecked code. The interaction between static types and Dyn is the same as normal gradual typing. However, the interaction between Untyped and any other kind of type, even Dyn, is not normally allowed. To interact, values need to be explicitly cast from or to Untyped. This interaction is made explicitly to prevent accidental sending of values from or to unchecked code. The explicit casts add all the required protection to checked values to prevent unauthorized changes of these values from unchecked code.

The example provided by Swamy for an usage of this trichotomy is in a gradually typed JavaScript with a focus in security. Statically typed code and checked dynamically typed code are code that is checked by the typechecker. Because of this, the typechecker can prevent bad program behavior (*e.g.* Changing the prototype of an unrelated object). Meanwhile, unchecked code correspond to code not checked by the typechecker, like third party libraries.

2.4 Performance related work

As a value flows in a program, casts can pile up. Different strategies exist to deal with chains of casts. They can be reduced as eagerly or as lazily as possible, yielding different flexibility/strictness tradeoffs [49]. However, even with an eager approach, higher-order casts, *i.e.* casts on function types, cannot be fully resolved eagerly and typically imply wrapping functions in proxies that perform casts upon entry and exit. As noted by Herman *et al.*, this approach can result in unbounded growth in the number of proxies, affecting both space efficiency and tail call optimization [30]. They propose the use of coercions instead of proxies to be able to combine adjacent coercions in order to limit space consumption. Going a step further, Siek and Wadler develop threesomes as a data structure and algorithm to represent and normalize coercions [52]. A threesome is a cast with three positions: source, target, and an intermediate lowest type. Combining a sequence of threesomes is done by taking the greatest lower bound of the intermediate types.

Chang *et al.* report on JIT-level optimizations based on optional type information [16]. They studied two groups of optimizations: low level and high level. Low level optimizations are optimizations that are language agnostic, optimizing the bytecode using the bytecode itself (*e.g.*, redundant store and load elimination, variable stored in a physical register). High level optimizations are aware of the language and runtime behavior (*e.g.*, method inlining, type inference). The conclusion they reached is that low level optimizations are not sufficient to pay back the time invested on them. However, introducing high level optimizations, the optimizations can increase performance by a significant amount. This work is however in ActionScript, which is not gradual in the sense of Siek and Taha in that it does not rely on the consistency relation and does not support higher-order casts.

Also in the context of ActionScript, Rastogi *et al.* use local type inference to eliminate occurrences of the dynamic type and therefore augment the "static-ness" of programs [44]. They report very encouraging results: on average, they observe a 1.6x improvement with inferred types, and up to 5x in certain cases.

The expected overhead of fine-grained integration between typed and untyped code that gradual typing supports has led several researchers to develop alternative ways to do the integration. In the Typed Racket language, the granularity of integration between statically typed and dynamically typed code is at the module level: a module is either typed or untyped, but it cannot mix both disciplines internally [57]. This reduces the flexibility of the integration somewhat, but also reduces the cases of interaction, while proposing a reasonable engineering trade-off. Interaction between typed and untyped code in Racket is mediated through contracts [58], with blame tracking.

2.5 Blame tracking

A transversal issue when dealing with casts is whether or not blame tracking is done in order to report the guilty party whenever a cast fails [22, 49, 62]. Siek *et al.* propose different strategies for blame assignment that may lead to blaming different parties for the same example [49]. These strategies are: Eager UD, Eager D, Lazy UD and Lazy D.They result from taking two decisions. The first decision is whether a location where a Dyn cast is realized could be blamed or not of a cast exception. An UD blame strategy is a strategy where casts from and to Dyn can be blamed for a cast exception. In contrast, a D blame strategy does not allow cast to Dyn to be blamed. The second decision is when a cast exception should be raised when a wrapper is added that will always raise a cast exception when used. An eager blame strategy will raise this exception when the wrapper is added, and a lazy blame strategy will raise this exception when the wrapped closure is used.

To illustrate the differences between blame strategies, we use the following code snippet:

```
1  a = <Dyn⇐String→Int> func
2  b = <Int→Int⇐Dyn> a
```

Using a lazy blame strategy, the example code would never raise an exception because b is never applied. In contrast, using an eager blame strategy will raise an exception when executing Line 2. The line that is blamed depends if the blame strategy is UD or D. With an UD strategy, the blame is assigned to Line 1, because the mismatch is at the argument side and in that case the culprit is the cast to Dyn. With a D strategy, the blame is assigned to Line 2 because that is where the cast from Dyn was done.

Adding blame tracking complicates matters, both theoretically and practically. For instance, threesomes with blame is considerably more complex to understand than threesomes without blame [52, 25].

2.6 Other type systems

Integrating static and dynamic type checking is a highly active area of research. Gradual typing is just one of several partial typing techniques developed in the literature. Some other popular partial typing techniques are soft typing, pluggable types, hybrid typing and like types. We discuss these here.

Soft typing [14] is a static type discipline that does not reject potentially erroneous programs, but inserts casts to ensure safety. A soft type system does not have type declarations. It uses an inferencer to add the type declarations. If it finds an inconsistency, the type system will warn the programmer and will add dynamic checks. In contrast, gradual typing allows the programmer to use type declarations and insert casts whenever there is an inconsistency with those declarations

Pluggable types [9] is an approach that allows plugging optional type systems into code. Because these optional type systems may interact with another optional type system, they cannot modify the compiled code in any form. However, this allows to plug in as many optional type systems as needed, without the risk of conflicts between them. Also, it allows to enable at discretion an optional type system only when it is needed, because the final compiled code is the same either way. This can be beneficial when the typechecking process is time consuming. In contrast, gradual typing modifies the compiled code to insert the casts. Because of this, gradual typing can give more guarantees compared with pluggable types, without sacrificing the seemless integration between static typing and dynamic typing.

Hydrid typing [35] combines static typing with refinement types [23]. An automated theorem prover is used to check type consistency, and run-time checks are inserted where static type inconsistencies cannot be detected. The main difference between hybrid typing and gradual typing is where those type inconsistencies cannot be detected. In Hybrid typing, undecidable type consistency arises because of complex contracts. Checking if any contract is a subtype of another contract is an undecidable problem. In gradual typing, undecidable type consistency practically arises because of the existence of the Dyn type. The reason is because type consistency in gradual typing is too permissive with Dyn types, making it necessary to check at runtime that the assumption made by type consistency is correct.

Wrigstad *et al.* propose another approach to alleviate the performance issue of gradual types, integrated in Thorn [7, 63]. Instead of relying on the type consistency relation, they introduce a novel intermediate point between dynamic and static types: like types. When a method argument is declared to be of a like type, its uses in the method body are statically checked. Clients of the method can pass any value, just as if there was no declared static type. Conformance is checked dynamically. Values typed with a like type permit all kinds of invocations, in the same way as the Dyn type. However, a like type declares a set of methods that the

	Type annotations	Casts	Static and dynamic integration
Soft types	no	yes	no
Pluggable types	yes	no	yes
Hybrid typing	yes	yes	no
Like types	yes	yes	limited
Gradual typing	yes	yes	yes

Table 2.1: Comparison of features between the different type systems

object must possess. All interactions between the static and dynamic world are done using like types. The Thorn compiler is able to aggressively optimize concrete types, and the authors report speedups of 2x to 4x between an untyped Thorn program and a fully typed one (with untyped libraries).

Table 2.1 presents a comparison of features between the different type systems. In general, all the type systems described here can be type annotated and use casts to check the integrity of operations when this is not possible at compile time. The exceptions are soft typing which does not accept type annotations at all, and pluggable types which cannot use casts. Gradual typing, like types and pluggable types provide an integration between static and dynamic typing, while hybrid typing and soft typing are completely statically typed. Moreover, like types provide this integration only where like types are used. We choose gradual typing for our research because it provides a full integration between static and dynamic typing and it also uses casts to check that static type assumptions are respected at runtime.

Chapter 3

Gradualtalk

This chapter[1] presents the design and implementation of Gradualtalk, a gradually-typed Smalltalk. Implementing Gradualtalk was the necessary step to finding and validating any proposal for improving the performance of gradual typing.

The validation of Gradualtalk itself is not the focus of this work. A fully explained validation of Gradualtalk can be found in the PhD thesis "Empirically-Driven Design and Implementation of Gradualtalk" [11]

3.1 Introduction

Designing and implementing a gradual type system for Smalltalk is a challenging task because of the highly-dynamic nature of the language and the "live" programming environment approach. Indeed, Smalltalk is a reflective language that cannot be, in general, easily typed. Moreover, incremental programming in Smalltalk implies accepting partially-defined methods by the type system and to dynamically (at runtime) react to class updates. Additionally, as in any language, programmers rely on various programming idioms, some of which are challenging to type properly. These Smalltalk particularities make the design and implementation of a gradual type system a challenge in itself.

This chapter reports on the design and implementation of Gradualtalk[2], a gradually-typed Smalltalk, which is fully compatible with existing code. Following

[1]This chapter is based on the publication: "Gradual Typing for Smalltalk" [2].

[2]Available online at `http://www.pleiad.cl/gradualtalk`

the philosophy of Typed Racket [57], a major design goal of Gradualtalk is that the type system should accommodate existing programming idioms in order to allow for an easy, incremental path from untyped to typed code. The design of Gradualtalk was guided by a study of existing Smalltalk projects, incrementally typing them in Gradualtalk.

The type system of Gradualtalk supports Smalltalk idioms as much as possible through a number of features: a combination of nominal and structural types, union types, self types, parametric polymorphism, and blame tracking, amongst others. While there is no groundbreaking type system feature in Gradualtalk, the combination is quite novel, and the choice of these features—as well as their interactions—is carefully discussed in this chapter.

This chapter is organized as follows. We introduce Smalltalk in Section 3.2. We then introduce Gradualtalk by examples in Section 3.3. We then refine it with several typing features in Section 3.4. Later, in Section 3.5, we explore subtyping and runtime coercion semantics. Section 3.6 discusses implementation challenges. We finally summarize in Section 3.8.

3.2 Introduction to Smalltalk

Smalltalk [26] is a dynamically typed, class based, object oriented language. This section introduces Smalltalk syntax and concepts that are used in the rest of the chapter. For readers that are familiar with Smalltalk, this section can be safely skipped.

3.2.1 Programming in Smalltalk

Messages and method declaration The following code snippet is the implementation of the method distanceTo: in the Point class that calculates the distance between two points.

```
Point≫ distanceTo: p
    "This method calculates the distance between two points"
  |dx dy|
  dx := self x − p x.
  dy := self y − p y.
  ^ (dx squared + dy squared) sqrt
```

The first line is the method declaration. Point is the name of the class where the method is defined, distanceTo: is the name of the method and p is the name of the first and only argument. The second line is a comment. Comments can be placed in any place in the method and they are enclosed by double quotes("). The third line declares the local variables dx and dy. The fourth line until the seventh line are the statements of the method. Statements are separated by periods. The last statement can end in a period, but this is optional.

The fourth line calculates the distance in the x axis between the two points and stores it in the local variable dx. Assignment is denoted by the operator :=. self is the object that is the receiver of the message. It is the equivalent of `this` in Java and C++. self x, p x and self x – p x are all message sends to objects. Because everything in Smalltalk is an object, nearly every computation is performed by sending a message to an object. There are three kinds of messages: unary, binary and keyword:

- Unary messages have only the receiver as its argument. self x and p x are unary messages that are sending the x message to the objects self and p respectively.
- Binary messages have one additional argument and can only use special characters (*e.g.* +–∗/@<>). self x – p x is a binary message, where the message – is sent to the object returned by self x with an additional argument: the value of p x.
- Keyword messages can have one or more additional arguments. p1 distanceTo: p2, which is an usage of the method that is being defined, is a keyword message. The colon is required for keyword messages and if two or more arguments are required, more keywords are needed. An example is 1 to: 10 by: 2, which creates a collection of integers from 1 to 10 with an increase of 2 for each element.

The precedence of messages is unary messages first, then binary messages and finally keyword messages. For same kind of message, left message have precedence over right message. For example, the value of 2 + 3 ∗ 5 is 25 and not 17, because

2 + 3 is evaluated first. Parenthesis can be used to change the precedence. After evaluating `self x – p x`, the result is stored in the variable `dx`.

The seventh line returns the value of `(dx squared + dy squared) sqrt`. Return is denoted by the operator ^. In the case that no value is explicitly returned by the programmer, the method will implicitly return `self` as a value. This guarantees that all methods return a value.

Control flow and blocks Smalltalk features a rich set of control flow mechanisms. The following examples are a small set of those:

- Conditional: `(p < 3) ifTrue:[a:=1] ifFalse:[a:=0]`
- While: `(p < 3) whileTrue: [p := p + 1]`
- For: `1 to: 10 do: [:i|acc := acc + i]`
- Iterator: `points do: [:p|dispersion := dispersion + p distanceTo: zeroPoint]`

As mentioned before, everything in Smalltalk is done by sending messages and the control flow mechanisms are no exception. However, to effectively use the control flow, it is necessary to use another feature of Smalltalk: blocks.

Blocks are closures and are defined by enclosing the list of instructions with square brackets([ ]). Blocks can also declare that they need arguments to be executed. The arguments are defined by a list of parameter name with a colon at the start of each name. The list of arguments is separated from the body of the block by a vertical line(|).

The value returned by executing the block is the value returned by the last statement of the block. If the return operator is executed in a block(^), the method where the block was defined will end its execution and return the value specified by the return operator.

Because blocks store their contexts, the instructions stored in them can access the variables that were available where it was defined, no matter where the block is being executed. In formal language theory, this kind of data is know as *closures*.

Constants Smalltalk has the following as constants:

- Integers: 42
- Floats: 3.14
- Booleans: true, false
- Characters: $c
- String: 'Hello world!'
- Symbol: #Point
- Array constructors: {p1 x. p2 x}
- Null object: nil

3.2.2 Classes and metaclasses

As said before, Smalltalk is an object oriented language with classes. The following code snippet is the declaration of class Point.

```
Object subclass: #Point
   instanceVariableNames: 'x y'
```

The first line declares that the name of the new class is Point and it is a subclass of Object. The second line declares two instance variables: x and y. There is a more complex version of the class declaration available, but it is not necessary for the understanding of this thesis. The class declaration above is also a message sent to an object: Object.

Everything in Smalltalk is an object. This even includes the classes. Classes are objects that create and instantiate objects. They also act as a dictionary of all the methods that are shared between the objects instantiated by themselves.

All objects have a class. In the case of classes, their class is known as a *metaclass*. The class of Point is Point class. Because metaclasses are also objects, they have a class. Their class is Metaclass. The class of Metaclass is the metaclass Metaclass class and the class of Metaclass class is Metaclass, closing the relationship.

If the class A is subclass of class B, then A class is subclass of B class. The only exception is the root of the hierarchy, Object[1]. Because Object is the root of the hierarchy, Object has no superclass. However, the superclass of Object class is Class. This is done for the purpose that all classes can understand the methods defined in Class. Both Class and Metaclass have a common ancestor, Behavior. The relationship described here between classes and metaclasses is represented in Figure 3.1.

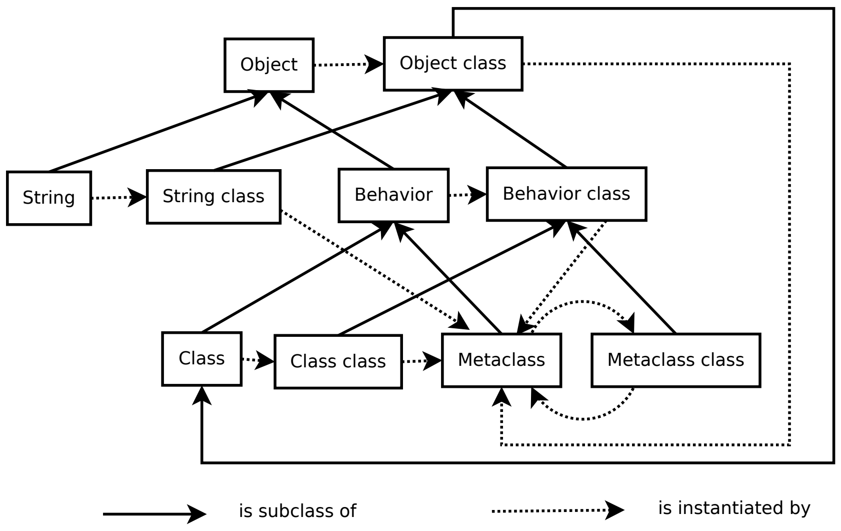

Figure 3.1: Relationship diagram between classes and metaclasses in Smalltalk.

3.3 Gradual Typing for Smalltalk

In this section, we present the Gradualtalk language, which is a Smalltalk dialect with gradual typing support. We now showcase the features of the language using as an example code snippets from a geometric calculation module.

[1]In Pharo Smalltalk, the real root of the hierarchy is ProtoObject, but we are ignoring that detail here.

3.3.1 From dynamically typed to gradually typed code

A developer is trusted with the development of the geometric calculation module for a graphics application. She starts writing dynamically-typed code. The following code snippets are the implementation of two example methods: euclidean distance and a class method for creating points.

```
Point≫ distanceTo: p
  |dx dy|
  dx := self x − p x.
  dy := self y − p y.
  ˆ (dx squared + dy squared) sqrt
```

```
Point class≫ x: aNumber1 y: aNumber2
   ˆself new x: aNumber1; y: aNumber2
```

After development and testing, the developer wants to increase robustness and provide basic (checked) documentation for these methods. For that purpose, she needs to type the method declarations of those methods. The following example is the typed version of the method distanceTo:.

```
Point≫ (Number) distanceTo: (Point) p
  |dx dy|
  dx := self x − p x.
  dy := self y − p y.
  ˆ (dx squared + dy squared) sqrt
```

The method declaration of this method specifies that the type of the parameter p is Point, while the return value type is Number. Because the local variables dx and dy are not annotated, they are treated as being of type Dyn, *i.e.* the type of any object in a dynamically-typed language.

Note that the Dyn type is also very helpful for typing methods that cannot otherwise be typed precisely, either because of a limitation of the type system, or because of inherent dynamicity. The typical example of the latter is reflective method invocation, done in Smalltalk with the perform: method:

```
Object >> (Dyn) perform: (Symbol)aSymbol
```

The argument to the method perform is a Symbol, which denotes the name of the method (selector) that must be invoked on the receiver object. In general, the return type cannot be statically determined. Declaring it as Dyn instead of Object means that clients of this method can then conveniently use the return value at any type, instead of having to manually coerce it.

3.3.2 Closures

The next method to type in our example is perimeter:. This method takes as parameter a closure that computes the distance between two points, and returns the value of the perimeter of the polygon, using the provided closure. Closures, also known as blocks, are a basic feature in Smalltalk, so the type system supports them. The following code is the typed version of the perimeter: method declaration:

```
Polygon ≫ (Number) perimeter: (Point Point → Number) metricBlock
   ...
```

In the example, the parameter metricBlock is a closure; its type annotation specifies that it receives two Points and returns a Number.

3.3.3 Self and metaclasses

The next method to type is y:. This method is a setter for the instance variable y. Its return value is self, the receiver of the message. The following code corresponds to its typed method implementation:

```
Point≫ (Self) y: (Number) aNumber
   y := aNumber.
```

Self is the type of self, as in the work of Saito *et al.* [47]. Declaring the return type to be Point would not be satisfactory: calling y: on an instance of a subclass of Point would lose type information and forbid chained invocations of subclass-specific methods.

We now consider the class method x:y:, which acts as a constructor:

```
Point class≫ (Self instance) x: (Number) aNumber1 y: (Number) aNumber2
   ^self new x: aNumber1; y: aNumber2
```

Self instance is the type of objects instantiated by self. Self instance is therefore only applicable when self is a class or metaclass. This was inspired by the type declaration “Instance” in Strongtalk [10]. Using Self instance instead of Point offers the same benefits as explained above. Constructor methods are inherited, and Self instance ensures that the returned object is seen as an object of the most precise type. The dual situation, where an object returns the class that instantiated it is dealt with using Self class, also inspired by Strongtalk.

Self instance in Gradualtalk and Strongtalk Instance are similar, but subtly different. The difference shows up when looking at the Class class, and related classes. Recall that in Smalltalk, classes are objects, instances of their respective metaclass, which derive from the Class class. The problem is that in Strongtalk, inside methods of that class, the type Instance is a synonym of Self. This means that all methods defined in Class—and its superclasses ClassDescription and Behavior—lack a way to refer to the type of their instances. This limitation can be observed in several places. For example, the return type of Behavior ≫ #new is Object in Strongtalk, which is imprecise, while it is Self instance in Gradualtalk. To type the method new correctly, Strongtalk needs to redefine new in the subclass Object class (the metaclass of Object), and change its return type to Instance. Another example of this problem is in the following method from Behavior:

```
Behavior≫ (Self) allInstancesDo: (Self instance → Object)aBlock
   "Evaluate the argument, aBlock, for each of the current instances of the  receiver."
```

Using Self instance above as the argument type of the block denotes any possible instance of a Behavior object. Properly typing this method is not possible in Strongtalk: as a consequence, it has been moved down the hierarchy to the Object class class. Self types in Gradualtalk are strictly more expressive than in Strongtalk.

3.3.4 Casts

The following code is the method perimeter, which computes the perimeter using the Euclidean metric:

```
Polygon ≫ perimeter
  ˆ self perimeter: [:x :y| x distanceTo: y]
```

This dynamically-typed method invokes the perimeter: method with a (Dyn Dyn → Dyn) closure, yet this method expects a (Point Point → Number) closure. In the Gradualtalk type system, the former closure type is *implicitly* cast to the latter. As a result, the developer does not need to write any type annotation.

The language also gives the programmer the option of explicitly coercing from one type to another type. An explicit cast is shown in the following:

```
Polygon ≫ perimeter
  ˆ self perimeter: [:x :y| (<Integer> x distanceTo: y)]
```

The return value of the expression “x distanceTo: y” is cast to an Integer. If it is not a subtype of Integer at runtime, a runtime exception is raised.

3.3.5 Blame tracking

Casts can fail. However, a higher-order cast (*i.e.* , a cast that involves function types or, by extension, structural object types) cannot be verified immediately and therefore this check must be delayed [22, 49]. This means that the point where an error is detected can be spatially and temporally decoupled from the actual source of the error. The general solution to this issue is to perform *blame tracking* [62]. When a check is delayed, the type system remembers information that will then allow blame to be properly assigned, pointing at the expression that is responsible for the error.

Consider the following:

```
PolygonTest ≫ testPolygon: b
    |block (Polygon)pol|
    block := (<Point Point → Integer> b).
    ...
    pol perimeter: block.
```

The cast of b to the proper function type cannot be checked immediately. Hence the type system ensures that clients of block use it properly by providing two Point arguments (otherwise they are to blame), and checks that the block always effectively returns an Integer (otherwise the cast itself is to blame, because it would have failed if we were able to check it immediately).

3.4 Refining the Type System

In this section, we extend the gradual type system introduced in Section 3.3. These extensions are a series of features that we found necessary while typing several Smalltalk projects. We conclude that these are necessary for a smooth migration of Smalltalk projects to a typed version that is as precise as possible. Indeed, since no type annotation means Dyn, any Smalltalk program is already a Gradualtalk program with only Dyn types. The objective is therefore to introduce some features that help minimize the number of required Dyn annotations in the code. As we show in this section, many Smalltalk programming idioms suggest a specific type system mechanism in order to avoid relying on Dyn.

Supporting programmer idioms is important for backward compatibility and seamless integration. The first simply is to maintain support for the legacy packages that are the core of today's Smalltalk programs. The second is because programmers should not have to refactor code to satisfy the type system.

3.4.1 Parametric Polymorphism

Consider the following piece of code, where an array of Dyn objects is defined:

```
|(Array) points|
...                    "filled with points"
(points at: 1) x       "potentially unsafe"
```

The programmer knows that any element of the array is a Point., and invokes the method x of class Point. Sadly, the type system cannot guarantee a safe method call at compile time. Consequently, a coercion is introduced by the type system. Here, the type information is lost, forcing the programmer to either use casts or the Dyn type. Casts need to be manually inserted, which is cumbersome and error prone.

To solve this problem, Gradualtalk supports parametric polymorphism [41]. Adding parametric polymorphism to gradually typed languages is not new: Ina and Igarashi [32] presented a formalization and initial implementation of generics for gradual typing in the context of Featherweight Java. We adopt their approach in Gradualtalk. As of now, generics are implemented using type erasure as in Java. That means that generic annotations are removed before execution time and the runtime does not require those annotations to work.

Gradualtalk includes a generically-typed version of the Collection library. For instance, the next piece of code solves the above problem by introducing Point as a type argument to the generic Array type:

```
|(Array<Point>) points|
...
(points at: 1) x       "safe call"
```

Below is an example of a generic method definition:

```
Collection<e> ≫ (a) add: (a <: e) newObject
```

This method inserts an object in a collection. Interestingly, in Smalltalk, the return value of this method is the added object. Therefore, in order to not lose type information, we use a bounded type variable a, subtype of the collection element type e, and specify a as the return type. Note that by convention, in Gradualtalk, type variables are single lowercase characters, similar to Haskell and ML.

Along with generics the type system also supports polymorphic functions (blocks in Smalltalk), which is useful in several cases, *e.g.*higher-order functions in collections:

```
Collection<e> ≫ (Self<f>) collect: (e → f) op
Collection<e> ≫ (Self<e>) select: (e → Boolean) pred
Collection<e> ≫ (f) inject: (f) init into: (f e → f) op
```

Note that the two first methods above use parametric self types to precisely type their return values.

Combining parametric types with some other typing features may produce new and interesting properties. For instance, the interactions between the Dyn type and generics, called bounded dynamic types [32], permits flexible bounded parametric types. Gradualtalk does not include this feature as of now. Another interesting interaction occurs between self types and generics, called self type constructors [47], allowing programmers to parametrize self types. Self type constructors are required to properly type collections in Gradualtalk, and are therefore supported.

3.4.2 Union Types

The following piece of code is a polymorphic implementation of the ifTrue:ifFalse: method, where we use RT as a placeholder for the return type:

```
Boolean≫ (RT) ifTrue: (→ a) trueBlock ifFalse: (→ b) falseBlock
  ...
```

The method receives two block arguments, one for the true case named trueBlock and one for the false case, named falseBlock. In this example each block can evaluate to a result with a different type. Because of this, the trueBlock has return type a and the falseBlock has b, and a=b is not always the case. Consequently, at this moment there are two possible values for RT:

Object. The type Object does not provide any information to programmers. Even if we consider the lowest common ancestor between types A and B, still some type information is lost. Therefore, programmers are forced to insert a cast to get the real type.

Dyn. We get the flexibility that we need, but again type information is lost.

While this is a simple example, there are several places where examples like this can be found. To solve this problem, we use *union types* [41]. These allow programmers to join several types in a single one, via disjunction. Union types are represented by | in Gradualtalk. A union between types a and b solves the problem of the example, letting the programmer specify that only one of these is possible.

```
Boolean≫ (a | b) ifTrue: (→ a) trueBlock ifFalse: (→ b) falseBlock
```

Another example is the following method:

```
Collection<e> ≫ (Self | a) ifEmpty: (→ a) aBlock
  ^ self isEmpty ifTrue: [ ^aBlock value ] ifFalse: [ self ]
```

The method returns the result of the invocation of aBlock (of type a) if the collection is empty, or self otherwise. To type this precisely, a union type Self | a is used.

When using a variable typed with a union type a | b, the programmer can safely call common methods in a and b. Calling specific methods of a or b requires explicit disambiguation, for instance using isKindOf: to perform a runtime type check and then using a coercion.

3.4.3 Structural Types

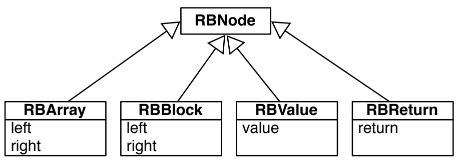

Figure 3.2: A common structural protocol.

Figure 3.2 describes the RBNode hierarchy (RB is shorthand for Refactoring Browser) that represents abstract syntax tree nodes in a Smalltalk program. In the example, only the classes RBArray and RBBlock understand the selectors left and right.

Consider the following code that is added to handle brackets in the parser, where we use AT as a placeholder for the argument type:

```
RBParser ≫ bracketsOfNode: ( AT ) node
... node left.
... node right.
```

Consequently, there are three possible values for AT:

RBNode. RBNode is the common ancestor of RBArray and RBBlock. However any call to the methods left and right will be rejected by the type system, because RBNode does not define these methods. Even a cast will not help, because the type system cannot statically determine if either RBArray or RBBlock will be the correct type.

RBArray | RBBlock A union type could be a good solution. However it is not scalable if more nodes include brackets later on in development.

Dyn. The code will be accepted by the type system, but again type information is lost.

This problem appears because RBArray and RBBlock have no relation between one another except that they are nodes, and not all nodes have brackets. But RBArray and RBBlock also share a set of common methods used in the method bracketsOfNode:. Therefore, objects of type AT will understand this set of methods, *i.e.* the selectors left and right. A type with this structural representation, *i.e.* set of method types, is called a *structural type* [41]. This means that the type system permits as argument any object that understands the selectors in the structural definition. Using structural types, the solution is as follows:

```
RBParser ≫ bracketsOfNode: ({left (→ Integer) . right (→ Integer)}) node
```

The syntax of structural types is {methodA (T_1) . methodB (T_2) . ...} where methodA and methodB are the names of the methods declared in the structural type and T_1 and T_2 are the types of the corresponding method as a function type.

The use of structural types allows programmers to explicitly specify a set of methods that an object must implement. These methods are the only available methods for the structurally-typed variable (node, in the above example) and therefore any call to another method will be invalid, unless a cast is used.

Type Alias The verbosity of structural types could be a problem for programmers, and even worse it can lead to an agglomeration of anonymous protocols. To solve this, Gradualtalk permits the use of a *type alias* [41], where programmers can give names to arbitrary types in order to enhance readability. Note that the use of a type alias is not only restricted to structural types. For example, Nil is a type alias for the UndefinedObject type.

Named Protocols Smalltalk does not support explicit interfaces or protocols. Instead, programmers rely on their understanding of what a given protocol is, and provide the necessary methods. For example consider the *pseudo* protocol "property", where the methods that handle properties in an object are listed:

```
propertyAt:
propertyAt:Put:
propertyAt:ifAbsent:
propertyAt:ifAbsentPut:
removeProperty:
removeProperty:ifAbsent:
```

Not making this protocol explicit is fragile, because it may evolve over time.

By combining a structural type and a type alias, programmers are able to define *named protocols*, which are similar to nominal interface types, except that they are checked structurally. With this, protocols are explicitly documented, and programmers can explicitly require them *e.g.* as an argument type of a method, without losing the flexibility of structural typing.

Note that a named protocol can work to give a type to a trait [48]. A trait is a set of methods that can be used to extend the functionality of a class. However, traits come with a specific implementation, while named protocols are pure interface specifications. The same protocol can be implemented by different traits.

3.4.4 Nominal Types

Nominal types [41] are the types that are induced by classes, *e.g.*, an instance of class String is of type String. One of the primary advantages of nominal types is that it helps programmers easily express and enforce design intent. Because of this, most mainstream statically typed object-oriented languages support nominal types rather than other alternatives, such as structural types.

Nevertheless, structural types offer their own advantages [37, 38]. For instance, structural types are flexible and compositional, providing better support for unanticipated reuse. This is because they imply a more flexible subtyping relationship compared to nominal subtyping, allowing unrelated classes in the class hierarchy to be subtypes. Taking this into account, some type systems [51, 20, 37] use structural types. In fact, a nominal type also can be considered in terms of its structural representation. This means that instances of class String have a structural type: the set of all methods that a string understands. Using such a type alias, programmers can benefit from the advantages of structural types.

In the case of Smalltalk, considering class-induced types as their structural representation is, however, not suitable. This is because Smalltalk classes tend to have a large number of methods, which makes it impractical to comply with subtyping outside of the inheritance hierarchy. For instance, consider the SequenceableCollection class, which has hundreds of methods. If the programmer wants to define a subtype that is not a subclass she must implement all methods in the SequenceableCollection class. A solution is to combine structural and nominal types, as discussed next.

3.4.5 Reconciling Nominal and Structural types

Figure 3.3 describes the hierarchy of some classes in Smalltalk that define the selectors left and right with type signature ($\rightarrow$ Integer). With this new set of classes, the solution presented in Section 3.4.3 is not complete. This is because the type system will accept calls to the method bracketsOfNode: with a parameter that complies with the protocol, *e.g.*, a Morph object, but which is not a node.

Gradualtalk supports the combination of nominal and structural types, similar to Unity [37] and Scala [19].[1] A type combines both a nominal part and a structural part, as in A{m1...mn}. For instance, consider the following modification in the parameter type that takes structural and nominal types into account:

[1]Note that because Scala compiles to the JVM, structural invocations introduce an extra performance penalty due to reflection. Gradualtalk, on the other hand, does not penalize structural invocations.

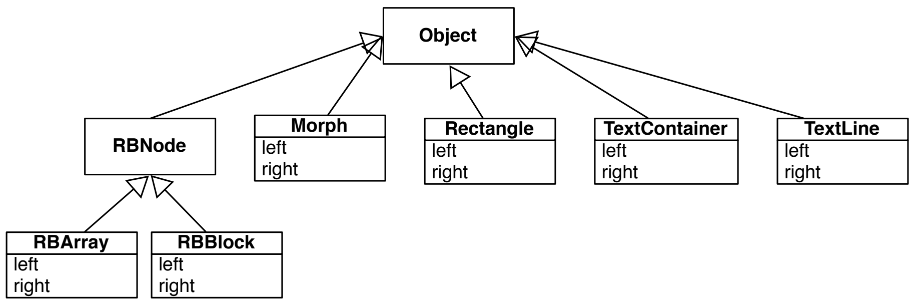

Figure 3.3: A common structural protocol across projects.

```
RBParser ≫ bracketsOfNode: (RBNode{left (→ Integer) . right (→ Integer)}) node
```

Here the type system is requesting an explicit RBNode object that has selectors left and right. Now a call with a Morph object as argument is rejected because it is not an RBNode.

Note that a nominal type A is a syntactic shortcut for the combined type A{} (empty structural component), while a structural type {m1, m2, ...} is the equivalent of Object{m1, m2, ...}.

Flexible Protocols Interestingly, the combination of the Dyn type with a structural type produces a *flexible protocol*, of the form Dyn {m1, m2, ...}. A flexible protocol represents objects that must comply with a protocol (structural part), but can otherwise be used with an implicit coercion (Dyn part). An object typed with flexible protocols allows the programmer to invoke any method. In contrast, an object typed with a structural type requires an explicit cast if the programmer wants to invoke any method not declared in its type. Assigning to a variable typed with flexible protocols will check that the object complies with the protocol. In contrast, any object can be assigned to a variable typed as Dyn.

Consider the following piece of code:

```
Canvas≫ (Self) drawPoint:(Dyn {x(→ Integer). y(→ Integer)}) point
... point x. "safe call"
... point y. "safe call"
... point z. "not an error, considering point as Dyn"
```

The last statement does not raise a type error, because point has been typed with a flexible protocol. However, calling drawPoint: with an argument that does not support the {x, y} protocol is a static type error. Since Unity and Scala are not gradually-typed, flexible protocols are a novel feature of Gradualtalk.

3.4.6 Safety and Type Soundness

Gradualtalk is based on Smalltalk, which is a *safe* language: sending an unknown message to an object is a trapped error that results in a MessageNotUnderstood exception, instead of producing unspecified result or system crash. Gradualtalk inherits this safety property.

With respect to type soundness, Gradualtalk follows the foundational work on gradual typing by Siek and Taha [51], with the blame assignment mechanism of Wadler and Findler [62]. The result is that Gradualtalk guarantees that, if a runtime type error occurs (that is, a MessageNotUnderstood exception is thrown), it is either due to an explicit cast that failed, or the consequence of passing an inappropriate untyped value to typed code. In the latter case, the error may occur anywhere in the code, but blame assignment will necessarily point to a faulty dynamically-typed expression that caused the error to occur later.

Note the practical value of blame assignment: without it, the programmer is left with the current call stack to investigate; however the root cause may be long gone and therefore not appear in the call stack.

3.4.7 Summary

τ	$::= \gamma \mid \overline{\tau} \rightarrow \tau \mid \tau + \tau \mid \gamma{<}\overline{\tau}{>} \mid \gamma\sigma$	Type
γ	$::= \nu \mid \epsilon \mid \mathsf{x} \mid \mathsf{Dyn}$	Ground type
ν	$::= \mathsf{C} \mid \mathsf{C\ class} \mid$	Nominal type
ϵ	$::= \mathsf{Self} \mid \mathsf{Self\ instance} \mid \mathsf{Self\ class}$	Self type
σ	$::= \{\overline{\mathsf{m}\ \overline{\tau} \rightarrow \tau}\}$	Structural type

Table 3.1: Types in Gradualtalk

Table 3.1 presents the grammar of types in Gradualtalk. C ranges over class names in the system, x ranges over type variables and m ranges over selector names. A bar over a type term specify zero or more occurrences of the term.

A type τ is a either a ground type γ, a function type, a union type, a generic type, or a combined type with a structural component σ. A ground type is either a nominal type ν, a self type ϵ, a type variable or Dyn. A structural type σ is a list of selector types, including a selector name and a function type.

3.5 Type System Semantics

We now describe three important aspects of the type system of Gradualtalk: self types, subtyping and runtime coercions.

3.5.1 Self types

Although the semantics of the type Self are well known, this is not the case for the Self instance and Self class types. To define them properly, we define the concepts of instance types and class types. The instance type of τ is the type of objects instantiated by objects of type τ. If an object of type τ cannot have instances, then the instance type of τ is undefined. For example, the instance type of String class is String. This is because the String class object (and its subclasses) instantiate objects of type String (or a subtype of it). The class type of τ is the type of the class object that produces objects of type τ. Figure 3.4 and Figure 3.5 define the rules for instance types and class types respectively. Note that a key challenge in Smalltalk is to properly take into account the core classes that describe classes and metaclasses: Behavior, its subclass ClassDescription, and its subclasses Class and Metaclass.

A self type can be found in a calling context, as the type of a parameter or return value of an invoked method, or in called context, as the type of a variable or of the return value. In a calling context, self types are replaced by the type of the receiver. If the type of the receiver of the invoked method is τ, Self, Self instance and Self class are replaced by τ, instance(τ) and class(τ) respectively. In a called context, the type Self is represented in the type system as $\mathsf{Self}_{\mathsf{C}}$, where C is the current class. Self instance and Self class are represented in the same manner.

$\text{instance}(\mathsf{Nil})=\mathsf{Nil}$

$\text{instance}(\mathsf{C\ class})=\mathsf{C}$

$\text{instance}(\mathsf{Metaclass})=\mathsf{Class}$

$\text{instance}(\mathsf{A})=\mathsf{Object}$, if $\mathsf{Class} <: \mathsf{A} <: \mathsf{Behavior}$

$\text{instance}(\mathsf{Self_C\ class})=\mathsf{Self_C}$

$\text{instance}(\mathsf{Self_C})=\mathsf{Self_C\ instance}$, if $\mathsf{C} <: \mathsf{Behavior}$

$\text{instance}(\mathsf{Self_C\ instance})= \text{instance}(\text{instance}(\mathsf{C}))$

$\text{instance}(\gamma\sigma)=\text{instance}(\gamma)$

$\text{instance}(\tau_1 + \tau_2)= \text{instance}(\tau_1) + \text{instance}(\tau_2)$

$\text{instance}(\mathsf{x})=\text{instance}(\text{upperbound}(\mathsf{x}))$

$\text{instance}(\gamma<\overline{\tau}>)=\text{instance}(\gamma)$

$\text{instance}(\mathsf{Dyn})=\mathsf{Dyn}$

Figure 3.4: Definition of the instance relation on types.

$\text{class}(\mathsf{Nil})=\mathsf{Nil}$

$\text{class}(\mathsf{Object})=\mathsf{Behavior}$

$\text{class}(\mathsf{C})=\mathsf{C\ class}$, if $\mathsf{C} \not<: \mathsf{Behavior} \wedge \mathsf{C} \neq \mathsf{Object}$

$\text{class}(\mathsf{A})=\mathsf{A}$, if $\mathsf{A} \in \{\mathsf{Behavior}, \mathsf{ClassDescription}\}$

$\text{class}(\mathsf{Class})=\mathsf{Metaclass}$

$\text{class}(\mathsf{Metaclass})=\mathsf{Metaclass\ class}$

$\text{class}(\mathsf{Self_C\ class})=\text{class}(\text{class}(C))$

$\text{class}(\mathsf{Self_C})=\mathsf{Self_C\ class}$

$\text{class}(\mathsf{Self_C\ instance})=\mathsf{Self_C}$

$\text{class}(\gamma\sigma)=\text{class}(\gamma)$

$\text{class}(\tau_1 + \tau_2)=\text{class}(\tau_1)+\text{class}(\tau_2)$

$\text{class}(\overline{\tau} \to \tau)=\mathsf{BlockClosure\ class}$

$\text{class}(\mathsf{x})=\text{class}(\text{upperbound}(\mathsf{x}))$

$\text{class}(\gamma<\overline{\tau}>)=\text{class}(\gamma)$

$\text{class}(\mathsf{Dyn})=\mathsf{Dyn}$

Figure 3.5: Definition of the class relation on types

3.5.2 Subtyping

One important feature in object-oriented languages is subtyping, by which an object of a given type can also be considered as being of any of its supertypes. The presence of several kind of types in Gradualtalk makes the subtyping relationship nontrivial. We next explain how it is treated.

Basic Forms of Subtyping Lambda types, self types, union types and parametric types have well-known subtyping relationships [41, 13, 47, 32]. Gradualtalk follows these rules. However, because of our extension to self types, there are two additional subtyping rules concerning self types:

$$\text{(Self instance)}\frac{}{\mathsf{Self_C\ instance} <: \text{instance}(\mathsf{C})}$$

$$\text{(Self class)}\frac{}{\mathsf{Self_C\ class} <: \text{class}(\mathsf{C})}$$

Bottom Type In Gradualtalk, Nil (which is an alias for UndefinedObject) serves as the bottom type. Since this type is a subtype of any other type, the programmer can use either nil or raise exceptions in any place where a typed object is expected.

Nominal and Structural Subtyping As explained in Section 3.4.4, Gradualtalk supports the combination of nominal and structural subtyping as in Scala. First, note that Gradualtalk (as most mainstream languages) equates nominal subtyping with the inheritance relationship. Subtyping of mixed types is described by the following rule:

$$\text{(Mixed)}\frac{\gamma_1 <: \gamma_2 \qquad \text{structural}(\gamma_1) \cup \sigma_1 <: \sigma_2}{\gamma_1\sigma_1 <: \gamma_2\sigma_2}$$

This rule states that a mixed type A {n1,...} is a subtype of B {m1,...} if and only if A is a nominal subtype of B and the *union* of {n1,...} and all the methods of A (*i.e.* , the structural view of A) is a structural subtype of {m1,...}. The reason that this rule adds all the methods of A when checking the structural part of the types is because A alone can be a subtype of B {m1,....}. The definition of structural(.) is direct and omitted here for brevity.

Consistent subtyping As explained in Section 2.2.2, Gradual typing extends traditional subtyping to *consistent subtyping* [51]. Consistent subtyping does not present any specific challenge with respect to the different kinds of types in Gradualtalk. An interesting case to mention though is that of flexible protocols, since these are a novelty of Gradualtalk. Recall that a flexible protocol is a type of the form Dyn {m1,...}, *i.e.* , a type that combines Dyn with a structural type. The consistent subtyping relation for flexible protocols is defined by the rules Mixed-Dyn1 and Mixed-Dyn2:

$$(\text{Mixed-Dyn1})\frac{\tau \lesssim \sigma}{\tau \lesssim \mathsf{Dyn}\ \sigma} \qquad (\text{Mixed-Dyn2})\frac{}{\mathsf{Dyn}\ \sigma \lesssim \tau}$$

Mixed-Dyn1 states that τ is a consistent subtype of $\mathsf{Dyn}\ \sigma$, if τ is a consistent subtype of σ. This rule explicitly makes τ comply with the structural part of the flexible protocol. Mixed-Dyn2 states that $\mathsf{Dyn}\ \sigma$ is a consistent subtype of any τ. Indeed, it is valid to pass a value of type $\mathsf{Dyn}\ \sigma$ anywhere, since this is already the case with Dyn alone. Interestingly, both rules Mixed-Dyn1 and Mixed-Dyn2 correspond to two of the basic rules of consistent subtyping, $\tau \lesssim \mathsf{Dyn}$ and $\mathsf{Dyn} \lesssim \tau$, generalized to mixed types. Both of these basic rules are obtained when σ is the empty structure.

Note that flexible protocols also enjoy a direct subtyping relation as defined by the following rule Mixed-Dyn-sub:

$$(\text{Mixed-Dyn-sub})\frac{\sigma_1 <: \sigma_2}{\mathsf{Dyn}\ \sigma_1 <: \mathsf{Dyn}\ \sigma_2}$$

Mixed-Dyn-sub states that $\mathsf{Dyn}\ \sigma_1$ is a subtype of $\mathsf{Dyn}\ \sigma_2$, if σ_1 is a subtype of σ_2. This rule is the generalization of the reflexive rule $\mathsf{Dyn} <: \mathsf{Dyn}$ to mixed types; that rule can be recovered by considering both σ_1 and σ_2 empty.

3.5.3 Runtime Coercion

Coercions are expression-level checkers introduced by the type system at compile time, seen in Section 3.3.4. There are two kinds of coercions: explicit and implicit. Explicit coercions are written by the programmer, while implicit coercions are introduced by the type system to guarantee soundness. Coercions are performed at runtime either nominally or structurally through subtyping.

Method invocation Implicit coercions on method arguments can be performed either on the call site when sending the message, or at the beginning of the execution of the called method. In Chapter 4, we explore this decision from the point of view focused on performance, and present a novel third way to perform cast insertion on method invocation.

Union types To reduce the number of coercions at runtime, in the presence of union types these are simplified as follows: An object declared as a union type is implicitly coerced only in the calls to valid methods (methods in the union) that are not present in the intersection between the structural representation of the types in the union. Consider the following simplified example:

```
|({m1,m2} | {m2,m3}) obj|
obj m2. "no coercion needed. m2 is present in both {m1,m2} and {m2,m3} "
(<{m3}>obj) m3. "manual coercion is needed. m3 is not present in {m1,m2}"
```

In this example, the call to m2 is safe, however the call to m3 needs a manual coercion.

Structural types Coercions to structural types can be performed in two ways:

Eager. These are performed when the code is executed, checking that the object with the structural type understands all messages that are defined in the type. One benefit of this is that it aims to express the users intention, because objects are forced to understand all methods specified by the programmer. However, it could reject some codes that will never fail.

Lazy. These are performed on demand, checking only the used methods when they are invoked. This is more flexible, because it is not necessary to comply with the full set of methods, only a subset. However, it may be harmful, because some objects can accidentally match this subset of methods.

The choice between these options is hard to make and may depend on the context where these objects or types are used. We have chosen to use eager coercions in order to maximize the benefits of using static types in the first place. If practical experience ends up requiring to use lazy coercions, it can be integrated in the language, offering the choice to the programmer.

3.6 Implementation

The implementation of Gradualtalk extends Pharo Smalltalk, by adding a gradual type system. This extension consists primarily of three components: the core, the typechecker and the type dictionary. The core allows for the representation of types and their relationships in Smalltalk. The typechecker is a pluggable extension to the Smalltalk compiler that verifies the correct typing of methods before compilation. The type dictionary is where type information is stored, *i.e.* , the typing of instance variables and methods for each class. The current implementation of Gradualtalk is focused on expressiveness and correctness.

In this section, we present the problems we encountered while developing Gradualtalk that we believe are important to consider when implementing a type system in Smalltalk or a language with similar features.

3.6.1 Live system

Nearly every Smalltalk environment is a live system. This means that the developer writes the code, runs it and debugs it in the same execution environment. To support this live environment, individual methods can be compiled and added to an existing class. This is in contrast to other languages where the smallest compilation unit is a class. This feature of a live environment raises three problems for a typechecker.

The first problem is the granularity of the compiling process. In Smalltalk, the compilation process is done per method, instead of per class. Traditionally, a type checker prevents compilation when type errors are found. But with such a fine-grained compilation process, the traditional approach does not work. For example, if a programmer needs to define two mutually-dependent methods, when the first method is defined, the typechecker cannot know if the second method referenced is going to be defined later. The error should however not block the programmer from keeping this as-yet-buggy method and then define the second method. The same situation happens when loading code, since code loading in Smalltalk is just a script adding definitions one-by-one. In order to address this issue, we decouple the typechecking process from the compiling process: Gradualtalk can compile methods with type errors. Errors are collected in a separate typing report window.

The second problem is that the work done by the typechecker can become obsolete when new methods are introduced or an old method is modified. For example, if the return type of a method is changed from Integer to String, all methods that invoke it can potentially become ill-typed. To solve this problem, we introduce a dependency tracking system based on Ghosts [12], which allows the type system to properly support partially-defined classes and circular dependencies. Undefined classes and methods that are referenced are considered ghost entities. This allows the type system to check for consistent usage of as-yet-undefined entities. Dependency tracking considers both defined entities and ghosts. Each time the programmer updates or deletes definitions, the dependency tracker notifies the type system of which methods must be checked again. In case the type system detects some type errors, it reports the exact points of failure. More precisely, the dependency tracking system records bi-directional references between *dependents* and *dependees*. These dependencies are updated whenever a method is type-checked, and whenever the format of a class definition (variables) changes. The result of this process is a dependency graph of dependent and dependee nodes. A dependent node is either a pair $(\mathit{class}, \mathit{selector})$, for dependent methods, or a pair $(\mathit{class}, \mathit{variable})$, for dependent instance or class variables. A dependee node is either a class, for type related dependencies, or a pair $(\mathit{class}, \mathit{selector})$ for method invocation dependencies. Whenever a dependee node is updated, all dependents are re-checked and re-compiled (necessary because implicit cast insertion may have to change).

The third problem occurs when compiling typed system code that is critical for the operation of the virtual machine. It is common that programmers commit errors when typing code, especially if the code was not developed by them. With normal code, it is not a problem that a method fails when compiling, or cast errors are raised when they are executed. However, in critical code for the VM, having cast errors is fatal. For example, if the default error handler raises a cast error, an infinite loop is produced and the system is unresponsive, making it impossible to use the debugger. To address this problem, in Gradualtalk runtime cast insertion and checking can be disabled or enabled at will. To gradually type important and critical system parts, we used this feature to first focus on debugging the cause of typecheck errors at compile time, then progress to runtime cast errors. Also, disabling runtime casts after a cast error is raised allows us to use the debugger without further interference of the type system.

Gradual or optional? Disabling runtime casts insertion was built in Gradualtalk to address the problem discussed above. Interestingly, it can also be used to make the type system of Gradualtalk an *optional* type system, just like that of Strongtalk. Moreover, because code instrumentation can be enabled or disabled at will, Gradualtalk allows optionally-typed and gradually-typed code to co-exist in the same system; a combination which, to the best of our knowledge, has not been explored so far.

3.6.2 The Untouchables

Smalltalk permits a programmer to modify existing classes, including the structure of their instance variables, with a few exceptions. The classes whose object instance structure cannot be touched are Class, CompiledMethod and any of their ancestors. The reason for this is fundamental: the VM needs to know statically how they are composed so it can carryout method lookup and execute Smalltalk code without relying on the method lookup being implemented as Smalltalk code. Because of this restriction, we need to put the type information of methods and instance variables outside of the class, in the type dictionary.

There is also a set of virtually untouchable methods, *e.g.* whileFalse: in BlockClosure. These methods can be modified by a developer, however, the VM ignores those changes, because the compiler optimizes all methods that invoke them with a sequence of label and jump bytecodes that implement the original behavior of these methods. Because of this, we cannot modify them to realize coercions on their parameters. Instead, we insert coercions at their call sites.

3.6.3 Fragile classes

The type system uses nominal subtyping, as described in Section 3.5.2. For testing if a class is a subclass of another one, we use the tools provided by the Smalltalk environment. However, there is a time where using those tools can be detrimental, such as when the structure of the class is changing. For example, when changing the structure of the class Point, the environment in that moment can either answer that ColorPoint is or is not one of its subclasses.

The problem is that when changing the class structure, a typecheck is *required* to be performed on all the methods of the class and its subclasses. This is needed to verify that when an instance variable is removed, it is unused in the class or subclasses.

The above is not the only part where classes are fragile. Interrupting the process of modification of a class can have far-reaching consequences. For example, if the typechecker throws a type error, ColorPoint can become a subclass of a pseudo-Point that appears exactly like the class Point, has the same name, but nonetheless is a different class. This effectively makes ColorPoint not a subtype of Point. The developer has no easy way to be aware of this without the knowledge that the class is corrupted.

Both of these problems can be solved with two actions: separating the typechecker from the compilation process when a class is being modified, and recording the structure of the actual class hierarchy. This allows simulation of the change on the copy of the structure, while performing the typecheck on that copy without changing the current environment. If the typecheck fails, the actual hierarchy remains unchanged.

3.6.4 Performance

The current implementation of Gradualtalk is focused on expressiveness and correctness. However, that does not mean that performance is not important. If the use a typed library incurs a high performance overhead, this is likely to discourage the adoption of static types.

Sadly, this is the case for the first implementation of Gradualtalk. Initially, we believed that this was because of a lack of polish in the implementation. However, we discovered that some of the performance issues where caused inherently by how Gradual Typing worked, at least using Siek-Taha design as direct implementation. Using this insight as an incentive for our research, we proposed two novel ideas which are discussed in the following chapters.

3.7 Related Work

Before diving into our contributions in the next chapters, we now review related practical partial type systems available today. At the end of this section, we mention other interesting type systems.

Strongtalk Strongtalk [10] is a well-known statically typed Smalltalk dialect that incorporates several typing features. The Strongtalk type system is optional: it does not guarantee that the assumptions made in statically typed code are respected at runtime. There are two major versions of Strongtalk. The first one relies on a structural type system using brands (named structural protocols). The second version abandons brands and uses declared relations to determine subtyping. The main reason reported by Bracha for this change is the fact that structural types do not appropriately express the intent of the programmer, and are difficult to read, especially when debugging [8]. Strongtalk supports parametric polymorphism and self types. In contrast to Gradualtalk, *"Strongtalk is not designed to type Smalltalk code without modifications"* [10]. Pegon [53] is a recent optional type system for Smalltalk, inspired by Strongtalk. It includes all typing features of Strongtalk (including the conflation of Instance and Self in Class and related classes that we discussed in Section 3.3.3), and adds type inference.

The objective of Gradualtalk is to be a gradually-typed language, while both Strongtalk and Pegon are optionally-typed: they do not enforce any guarantees at runtime about the types of values [9]. However, as we discussed earlier, Gradualtalk can also be used like an optional type system, by deactivating the runtime casts insertion and checking. In addition, there are subtle differences: in Gradualtalk nominal subtyping is equated to inheritance (Section 3.5.2), the semantics of self types is refined (Section 3.3.3), and protocols are implicit (Section 3.4.3).

Typed Racket Typed Racket [60] is an extension to Racket in order to support statically typed Racket programs. Typed Racket provides smooth and sound interoperability with untyped Racket [57], by using contracts at the boundaries. Gradualtalk was directly inspired by Typed Racket in the sense that the type system should be flexible enough to support existing programmer idioms. Typed

Racket includes several interesting features, such as union types, occurrence typing, first-class polymorphic functions and local type inference. It is designed for the functional core of the Racket language.

The most notable difference between Gradualtalk and Typed Racket is the granularity of typed and untyped code boundaries. In Typed Racket, the boundary is at the module level: a whole module is either entirely typed or not at all. Expression-level boundaries are more costly but more flexible, in that it is possible to statically type portions of a class while leaving a few difficult expressions typed dynamically. Accordingly to this design philosophy, Typed Racket does not support explicit type casts. The limitations of the per-module approach have been reported by Figueroa *et al.* in an experiment to implement a monadic aspect weaver in Typed Racket; they had to resort to the top type Any everywhere in their system [21].

DRuby DRuby [24] is an optional static type system for Ruby, which uses type inference. Programmers can annotate their code, such as methods, and DRuby will checks these annotations using runtime contracts on suspicious code, *i.e.* DRuby infers type to discard well-typed code. DRuby includes union and intersection types, structural types (called object types), parametric polymorphism and self types, among others. Furthermore, DRuby introduces a novel dynamic analysis to infer types in highly dynamic language constructors, *i.e.* the use of eval, send and missing_method functions.

Although the type systems of Gradualtalk and DRuby are very similar, despite the language differences, there are some notable differences. Gradualtalk does not infer types (yet). Dynamic language operators, such as send (perform in Smalltalk), are dealt with using Dyn in Gradualtalk. While Ruby has proper classes as objects and class methods, DRuby does not support the notions of Self instance and Self class. This means that constructor (class) methods cannot be precisely typed, nor can the uses of class. Finally, DRuby is not a partial type system, so not all Ruby programs are valid DRuby programs.

Other type systems The previous type systems are just a few examples of types systems for dynamically-typed languages. There are several other proposals for Smalltalk [33, 34, 27, 29, 42], although Strongtalk is the most representative and complete. Some [33, 34, 27] are prior to Strongtalk, and they do not include all necessary features to type Smalltalk programs, as Strongtalk does. Haldiman *et al.* [29] present a practical approach to pluggable types implemented in Smalltalk where only code that contains partial type annotations are type checked using type inference and traditional static type checking. Finally, Pluquet *et al.* [42], present RoelTyper, a type reconstruction tool for Smalltalk that infer possible nominal types for variables (instance, temporary and argument variables, as well as return method) with an accuracy of 75%.

In addition, type systems have been developed for other dynamic languages, for instance for Python, JavaScript and ActionScript. RPython (Restricted Python) [5] is a statically-typed *subset* of Python, in which some dynamic features (*e.g.*dynamic modifications of classes and methods) have been removed. JS_0 [6] is a statically-typed version of JavaScript with inference, where both dynamic addition of fields and method updating are supported. ActionScript is one of the first languages used in the industry to embrace gradual typing, and efforts have been made to optimize it using local type inference [44]. Gradualtalk could certainly benefit from this technique.

3.8 Conclusion

Providing a practical gradual type system that supports programmer idioms in a highly dynamic language such as Smalltalk is a complex task. This requires meeting two goals: a type system design that properly supports common programmer idioms without unduly increasing its complexity and dealing with implementation tradeoffs for such a type system in a live environment.

In this chapter, we have introduced Gradualtalk, a practical gradually-typed Smalltalk that successfully meets the above challenges. The type system of Gradualtalk combines several state-of-the-art features, such as gradual typing, unified nominal and structural subtyping, self type constructors for metaclasses, and blame tracking. Gradualtalk is designed to ease the migration of existing, untyped Smalltalk to typed Gradualtalk code.

In the following chapters, using Gradualtalk, we partially address two issues raised from both the implementation and practical experiences: performance and reliability of static code. In Chapter 4, we explore different ways to insert casts in method invocations and show how it could improve performance. In Chapter 5, we define qualifiers that improve the reliability of static code, as well as indirectly improving performance.

Chapter 4

Cast Insertion Strategies for Gradually-Typed Objects

In the previous chapter, we presented the design and implementation of Gradualtalk, a gradually typed Smalltalk. In the process, two issues were highlighted: performance and reliability. In this chapter [1] we present different cast insertion strategies and how they can affect performance.

4.1 Introduction

As Section 2.2.4 described, the semantics of a gradually-typed language is typically given by translation to an intermediate language with casts, *i.e.* , runtime type checks that control the boundaries between typed and untyped code. A major challenge in the adoption of gradually-typed languages is the cost of these casts, especially in a higher-order setting. Theoretical approaches have been developed to tackle the space dimension [30, 52], but execution time is also an issue. This has led certain languages to favor a coarse-grained integration of typed and untyped code [59] or to consider a weaker form of integration that avoids costly casts [63]. Other approaches include the work of Rastogi et al. [44], using local type inference to significantly reduce the number of casts that are required.

[1] This chapter is based on the publication: "Cast Insertion Strategies for Gradually-Typed Objects" [4].

While developing Gradualtalk[1], a gradually-typed Smalltalk, our first concern was the design of the gradual type system, with its various features (Chapter 3). However, after addressing that concern, our focus shifted to the efficiency of casts, especially those related to method invocations. This is because method invocations are naturally very frequent in object-oriented programs, especially in pure object-oriented languages like Smalltalk. Casts incur a runtime cost, and we are interested in their efficiency so as to achieve an acceptable level of performance without losing the features of gradual typing. In the foundational paper on gradually-typed objects [51], Siek and Taha describe the semantics of cast insertion using a caller-side strategy—which we term the *call strategy*. Due to implementation issues (which have since been resolved), our very first implementation of cast insertion, before implementing the Siek-Taha approach, was however based on a different approach, which we name the *execution strategy*. Here, casts are inserted on the callee side, at the beginning of each typed method. Studying the performance of both approaches revealed that they have complementary strengths, and that a third approach, which we call the *hybrid strategy*, could combine the best of both approaches.

This chapter reports on the study of these three cast insertion strategies in Gradualtalk. We present the experimental setting for microbenchmarks in Section 4.2. We then describe all three strategies in turn in Sections 4.3, 4.4, and 4.5. In each of these sections, we informally describe the approach and report on microbenchmarks. We then report on macrobenchmarks in Section 4.6. We discuss memory consumption and other considerations in Section 4.7 and Section 4.8 concludes.

4.2 Experimental Setting: Microbenchmarks

We evaluate cast insertion strategies by implementing them for Gradualtalk, presented in Chapter 3. In Gradualtalk code, omitting type annotations is equal to specifying the unknown type, which is denoted as `Dyn`. Gradualtalk is currently implemented in Pharo Smalltalk version 2.0. Note that Pharo uses the Cog VM, which features a JIT compiler.

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

```
MyCollection ≫ (Self) addElement: (Integer)x
  collection addLast: x.

MyCollection ≫ (Integer) at: (Integer)index
  ^collection at: index.

MyCollection class ≫ (Self instance) new: (Integer)size
  ^super new
    collection: (OrderedCollection new: size);
    yourself.
```

Listing 4.1: MyCollection class.

Because there is no standard benchmark suite for Smalltalk, we designed both micro- and macrobenchmarks. We describe microbenchmarks in this section; they will be used in the explanation of the different strategies to give a first assessment of their performance. The macrobenchmarks are described in Section 4.6 to provide an evaluation of the performance of cast insertion strategies on larger-scale, real-world scenarios. To favor reproducibility, the Smalltalk image used to perform all our experiments can be downloaded from: http://pleiad.cl/gradualtalk/strategies

Microbenchmarks

We designed the microbenchmark setting to study the specific cost of each cast insertion strategy in both their best and worst cases.

To do so, we start with a typed collection, MyCollection, shown in Listing 4.1. Note that the addElement: method returns Self, following the convention for side-effecting methods in Smalltalk. The class method new: has Self instance as the return type, specifying that it returns an instance of itself.

We then use two versions of a client: untyped and typed. The client repeatedly inserts elements and then looks for them. The idea behind the microbenchmark is that it simulates two common actions done in programs: inserting elements and a search in an unordered collection. We used integers as elements because they are one of the most common data type used. Technically, the collection has sorted elements, but the search does not use that knowledge. We did not use random integers because we wanted to control where the search stopped and we consider that for our benchmarks, there would not be a difference between using

```
Client ≫ untypedClient: stop withSize: size
  |col|
  col := MyCollection new: size.

  1 to: size do: [:i|
    col addElement: i.
  ].
  1 to: size do: [:i|
    ((col at: i) > stop) ifTrue: [ ˆi ]
  ].
  ˆ−1
```

Listing 4.2: Untyped client.

```
Client ≫ (Integer) typedClient: (Integer)stop withSize: (Integer)size
  |(MyCollection)col|
  col := MyCollection new: size.

  1 to: size do: [:(Integer)i|
    col addElement: i.
  ].
  1 to: size do: [:(Integer)i|
    ((col at: i) > stop) ifTrue: [ ˆi ]
  ].
  ˆ−1
```

Listing 4.3: Typed client.

different integer collections of the same size. The code for both versions is shown in Listing 4.2 and Listing 4.3, respectively. The goal of the microbenchmarks is to measure the cost of each strategy for the calls that perform element insertion (#addElement:) and the calls that perform lookup of an element (#at:). Beyond these calls, the code run in each version is the same; in particular, methods #to:do:, #ifTrue: and #> are primitives, *i.e.* they are not subject to cast insertion.

We execute the microbenchmarks for different sizes between 1,000,000 and 10,000,000 elements. For each size, the experiment is repeated thirty times and the average time is calculated. The maximum measured relative error in the microbenchmarks is ±1.23%, with a confidence level of 95%. In each case, the value of stop is half that in size. The benchmarks were run on a machine with an Intel Core i7 3.20 GHz CPU, 4 GB RAM and 250 GB SSD disk, running Ubuntu 12.10.

4.3 Call strategy

The call strategy is the direct implementation of the specification of Siek and Taha [51].

Before explaining the informal description of the call strategy, we are going to briefly describe the concept of consistent subtyping, previously explained in Section 2.2.2. Consistency, denoted as $\sim$, is a relation that accounts for the presence of Dyn: Dyn is consistent with any other type and any type is consistent with itself. The consistency relation is not transitive in order to avoid collapsing the type relation [50]. A type σ is a consistent subtype of τ, noted $\sigma \lesssim \tau$, iff either $\sigma <: \sigma'$ and $\sigma' \sim \tau$ for some σ', or $\sigma \sim \sigma''$ and $\sigma'' <: \tau$ for some σ''.

4.3.1 Description

Essentially, the call strategy inserts casts at call sites whenever needed. A typed callee can therefore rely upon the fact that all callers have been checked previously, and hence assume that its arguments are of the proper types.

The call strategy has two different scenarios for inserting casts, depending on whether the receiver type in the call site is known at compile time. For call sites where the receiver type is known at compile time (*i.e.* non-Dyn), the compiler compares the type of the passed arguments in the invocation with the type of the parameters in the method declaration. If the argument type is not a subtype of the parameter type but it is a consistent subtype, then a cast to the parameter type is inserted.

For call sites where the receiver type is unknown at compile time (*i.e.* Dyn), the compiler inserts code that, at runtime, looks up the method type information and casts each argument to the expected parameter type. This lookup must take into account that methods can be overridden, and that the required type information is obtained from the appropriate overriding method. This lookup procedure is similar to the lookup used to retrieve a method when it is invoked. That means that when carrying out method lookup, the runtime could also retrieve the type information of that method for the usage of the cast strategy. However, this would require modifying the VM.

For example, in the case of typedClient, because col is statically known to be of type MyCollection and the argument i of type Integer, there is no cast inserted when adding an element to the collection:

```
...
col addElement: i.
...
```

Similarly, there is no cast needed when accessing the collection with at:.

In the case of untypedClient, however, there is no static type information available, therefore the call strategy inserts code that will, at runtime, retrieve the actual type of the receiver, and assuming it is statically typed (as MyCollection is), perform runtime checks on the arguments to ensure they match.

The code below shows the transformation of the call strategy in the untypedClient when addElement: is invoked (optimized for the single parameter case):

```
...
__rcv := col.
__typeParam := GTRuntime getTypeParamOf: #addElement:
                                     in: __rcv.
__rcv addElement: (<__typeParam>i).
...
```

Note that retrieving the method type information dynamically is costly; our current implementation maintains a cache per class that associates each selector with its argument types. As a matter of fact, if gradual typing were integrated at the virtual machine level, we could extend the existing infrastructure of polymorphic inline caches to deal with the type information, and hence further reduce the associated cost.[1]

4.3.2 Microbenchmarks

Figure 4.1 shows the results of the microbenchmark for the call strategy. Detailed numbers are in Table 4.1. We can observe that with a typed client, the call strategy exhibits almost identical performance as base Smalltalk, *e.g.*, 0.62 seconds versus 0.63 seconds for 10M elements. With the untyped code, the call strategy however takes up to 90 times the time of base Smalltalk: 57.49 seconds versus 0.63 seconds for 10M elements. This result reflects the fact that with a typed client, the call

[1]For simplicity, we focus on argument types only; in the three strategies, casts on return types are performed in the callee, following [51].

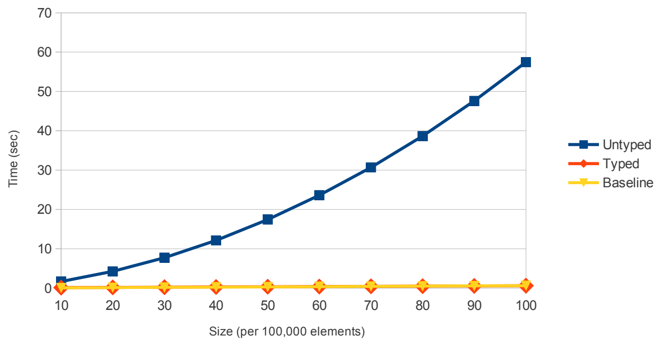

Figure 4.1: Running times of the call strategy for untyped and typed code, as well as of baseline Smalltalk.

Size (1M)	Call (untyped) (sec)	Call (typed) (sec)	Base ST (sec)
1	1.666	0.126	0.063
2	4.256	0.180	0.127
3	7.740	0.235	0.190
4	12.129	0.289	0.252
5	17.425	0.344	0.315
6	23.611	0.397	0.378
7	30.693	0.452	0.441
8	38.656	0.504	0.504
9	47.592	0.565	0.566
10	57.464	0.620	0.629

Table 4.1: Running times of the call strategy microbenchmark.

strategy does not insert any cast, making the resulting bytecode exactly the same as that of base Smalltalk. Conversely, with an untyped client, the call strategy inserts costly casts. This cost is slightly non linear, because of the increased usage of the garbage collector to recollect the objects used in the cast. The results show that even with the cache, the incurred overhead is substantial.

4.4 Execution strategy

As we have seen, the call strategy performs very well when a typed client calls a typed library, but it does not perform well when the client is untyped and the library is typed. This is unfortunate, because the scenario of a typed library that is used from untyped code is a predictably frequent scenario in Gradualtalk. Indeed, only a handful of libraries have been typed so far [2]. If using a typed library incurs a high performance overhead, this is likely to discourage the adoption of static types.

As it turns out, when first implementing Gradualtalk, a limitation of the compiler (which was subsequently resolved) prevented us from adopting the call strategy for cast insertion at first. To address this, we developed another approach, called the *execution strategy*, which turns out to perform well in the case of dynamically-typed receivers.

4.4.1 Description

The idea of the execution strategy is to insert casts on arguments of a statically-typed method directly at the beginning of the method. The useful characteristic of this strategy is that, in the case of a dynamically-typed receiver, there is no need to retrieve its type information at runtime.

In the execution strategy, the compiler inserts one cast per parameter at the start of the method. Each cast checks that the type of the value bound to the parameter corresponds with the declared type in the method signature.

The difference between the execution strategy and the call strategy is where the casts for the arguments are inserted. The call strategy inserts those calls just before the invocation of the method, while the execution strategy inserts the casts just after the method has been invoked.

For example, at the start of the method MyCollection >> #addElement:, the execution strategy inserts casts to the parameters of the method:

```
MyCollection ≫ addElement: x
  (<Integer>) x.
  collection addLast: x.
```

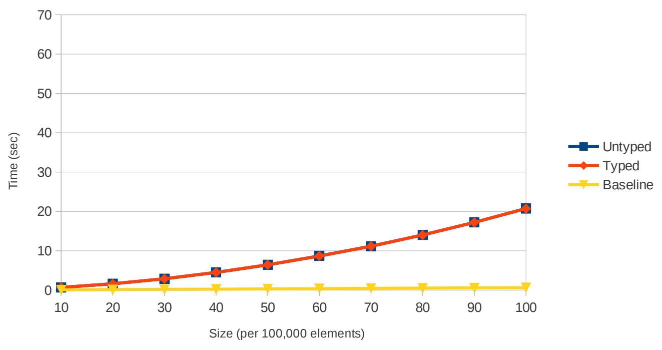

Figure 4.2: Running times of the execution strategy for untyped and typed code, as well as of baseline Smalltalk.

At call sites, regardless of whether the receiver is statically or dynamically typed, the execution strategy does not perform any transformation:

```
...
col addElement: i.
...
```

Note that this strategy is only meaningful in safe languages like Smalltalk, in which the runtime type of an object can be retrieved directly, *e.g.* through its class pointer. In the core semantics of gradually-typed objects of Siek and Taha, two-position casts play the role of tagging values with their type, so that injecting a value into Dyn does not lose its original type. This being said, all existing dynamic object-oriented languages that we are aware of are safe, and therefore the execution strategy is a meaningful option in all these languages.

Finally, observe that if casts are not implemented as primitives of the VM, then care must be taken to not create an infinite loop when the casting method uses methods of the system. This infinite loop can be produced if the casting procedure calls a method with at least one argument. In that case, because of how the execution strategy works, a cast would be done again in that argument when the method is invoked, creating the infinite loop. To avoid this, we have chosen to disable casts while a cast method is being executed, as it is the most straightforward approach.

Size (1M)	Call (sec)	Exec (sec)	Hybrid (sec)	Base ST (sec)	Exec vs Call (%)	Hybrid vs Exec (%)
1	1.666	0.671	0.626	0.063	-59.75	-6.71
2	4.256	1.609	1.529	0.127	-62.19	-4.97
3	7.740	2.899	2.785	0.190	-62.55	-3.93
4	12.129	4.504	4.357	0.252	-62.87	-3.26
5	17.425	6.433	6.252	0.315	-63.08	-2.82
6	23.611	8.705	8.473	0.378	-63.13	-2.67
7	30.693	11.158	11.010	0.441	-63.65	-1.32
8	38.656	14.033	13.870	0.504	-63.70	-1.16
9	47.592	17.222	17.054	0.566	-63.81	-0.98
10	57.464	20.748	20.528	0.629	-63.89	-1.06

Table 4.2: Running times of the untyped client microbenchmark.

4.4.2 Microbenchmarks

Figure 4.2 shows the results of the microbenchmark for the execution strategy. Detailed numbers for all microbenchmarks are in Tables 4.2 and 4.3. We observe that the execution strategy is unaffected if the client is typed or not: for 10M elements it takes 20.75 seconds with the untyped client, and 20.73 seconds with the typed client. This is because argument casts are inserted at the start of the methods of MyCollection, and are therefore always executed.

As a result, the call strategy is much faster than the execution strategy in the case of the typed client (around 93%). Recall that with the typed client, the call strategy does not insert any cast at all. Conversely, the execution strategy is considerably faster than the call strategy when using an untyped client (around 63%). While both approaches incur the cost of argument casts, the call strategy is slower because it first needs to retrieve the method type information (from the cache) to determine the actual cast to perform. In the execution strategy, the expected argument type is statically known, so only the proper cast is performed.

4.5 Hybrid strategy

The comparison of the call and execution strategies shows that they have complementary benefits. The call strategy performs well with typed clients, and the

Size (1M)	Call (sec)	Exec (sec)	Hybrid (sec)	Base ST (sec)	Call vs Exec (%)	Hybrid vs Call (%)
1	0.126	0.663	0.125	0.063	-80.95	-0.79
2	0.180	1.606	0.180	0.127	-88.81	-0.11
3	0.235	2.898	0.234	0.190	-91.89	-0.34
4	0.289	4.500	0.292	0.252	-93.59	1.18
5	0.344	6.430	0.346	0.315	-94.65	0.70
6	0.397	8.667	0.402	0.378	-95.42	1.31
7	0.452	11.146	0.455	0.441	-95.95	0.80
8	0.504	14.018	0.514	0.504	-96.40	1.94
9	0.565	17.212	0.564	0.566	-96.72	-0.11
10	0.620	20.725	0.620	0.629	-97.01	-0.13

Table 4.3: Running times of the typed client microbenchmark.

execution strategy performs well with untyped clients. We now present a novel strategy that combines the best of both strategies.

4.5.1 Description

The idea of the hybrid strategy is to trade space for speed. This is done by duplicating each method: one version is the original method that does not cast its arguments—called the *unguarded* method; and the other method starts by casting its arguments, as in the execution strategy—called the *guarded* method. Then, as in the call strategy, it has two different scenarios for inserting casts, depending on whether the receiver type in the call site is known at compile time or not. For statically-typed receivers, the compiler modifies the call site to invoke the unguarded method. If the argument type is not a subtype of the parameter type of the method declaration, the compiler inserts casts to the argument types. For dynamically-typed receivers, the compiler leaves the call site intact, as it will call the guarded method. If casts are also not implemented as primitives, then the same precaution with respect to the infinite loops made in the execution strategy should be taken in the hybrid strategy.

For instance, the addElement: method of MyCollection is replaced with the two methods:

```
MyCollection ≫ addElement: x     "guarded"
```

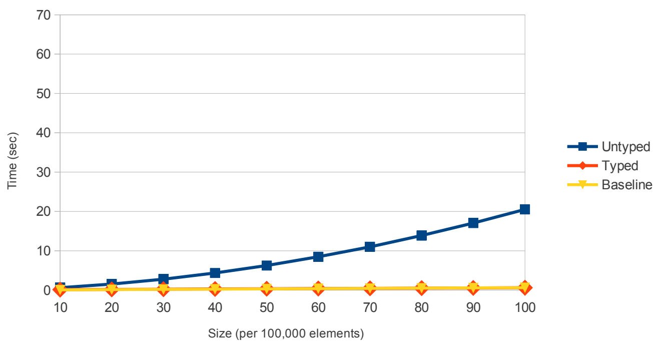

Figure 4.3: Running times of the hybrid strategy for untyped and typed code, as well as of baseline Smalltalk.

```
  (<Integer>)x.
  collection ___addLast: x.

MyCollection ≫ ___addElement: x  "unguarded"
  collection ___addLast: x.
```

When sending a message, if the type of the receiver is known, the code is modified to directly invoke the unguarded method:

```
  ...
  col ___addElement: i.
  ...
```

Otherwise, no transformation occurs, and hence the guarded method is called:

```
  ...
  col addElement: i.
  ...
```

4.5.2 Microbenchmarks

Figure 4.3 shows the results of the microbenchmark for the hybrid strategy. We can observe that the typed client runs faster than the untyped one: for 10M elements, it takes 20.53 seconds with the untyped client, and 0.62 seconds with the typed client. The difference reflects the code of the casts that are done repeatedly in the case of the untyped client.

We now compare the three strategies first on the untyped client and then on the typed client. In Figure 4.4 we show the execution times of the three cast

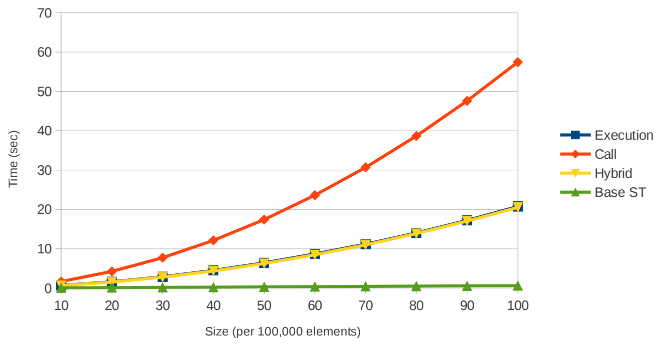

Figure 4.4: Running times of different cast strategies on the microbenchmark using the untyped client.

insertion strategies for the microbenchmark using the untyped client. The figure shows that for untyped code the call strategy is by far the slowest strategy, taking 57.46 seconds for 10M elements. The execution and hybrid strategies have what amounts to the same level of performance: for 10M elements 20.75 seconds and 20.53 seconds, respectively. These however still take up to 33 times longer than that of the time of base Smalltalk, which takes 0.63 seconds for 10M elements. The reason for this slowdown is that each strategy needs to perform some runtime casts, while the standard Smalltalk does not.

In Figure 4.5 we show the execution times of the three cast insertion strategies for the microbenchmark using the typed client. As can be expected, the execution strategy is the slowest, taking 20.73 seconds for 10M elements. The call and hybrid strategies have the same performance, and are nominally as fast as base Smalltalk (0.63 seconds for base Smalltalk and 0.62 seconds for both the hybrid and call strategy). This remarkable similarity is because none of these strategies need to do any cast. The execution strategy does perform such casts, which causes it to have a lower performance.

To conclude, the microbenchmarks confirm that the hybrid strategy performs as good as its best competitor in all cases.

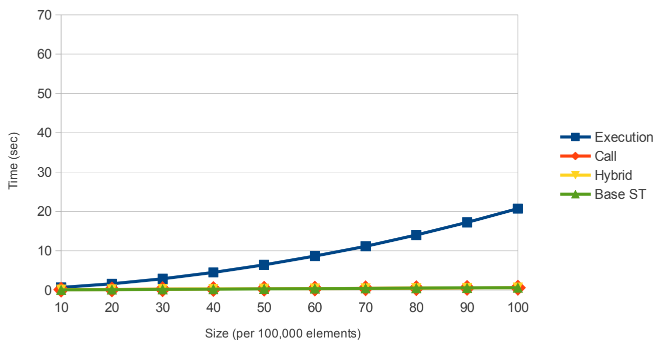

Figure 4.5: Running times of different cast strategies on the microbenchmark using the typed client.

4.6 Macrobenchmarks

In order to get an indication of whether the microbenchmark results carry over to larger-scale and real-world scenarios, we designed and performed some initial macrobenchmarks.

We have seen in the microbenchmarks that the call strategy is two orders of magnitude slower with untyped code. Considering that Smalltalk code is mostly untyped, the call strategy will be clearly outperformed in orders of magnitude by the execution and hybrid strategies on macrobenchmarks. Therefore, in this section, we only compare the performance of the execution and hybrid strategies.

4.6.1 Experimental Setup

We start with two scenarios that use untyped libraries:

- GZip: Compressing a 32MB XML file (adapted from a game project) to a gzip file. This is a writing I/O intensive operation.
- SAX: Parsing the same XML file with a SAX parser. This is a reading I/O intensive operation.

In order to measure how both strategies perform when type annotations are added, we developed partially-typed variants of these libraries. GZip-T adds some annotations to the GZip library. And for SAX, we developed two partially-typed

	GZip-T	SAX-T1	SAX-T2
typed classes	1	2	3
methods	23	62	116
fully typed	2	31	51
untyped	21	29	69
partially typed	0	2	3

Table 4.4: Characterization of the partially-typed libraries.

versions, SAX-T1 with few type annotations, and SAX-T2 with more annotations. Note that even though the code in the image has no type annotations, some types are implicitly deduced: self and literal values.

To characterize the extent to which each library is typed, we classify methods in three categories: typed methods, partially-typed methods and untyped methods. A fully-typed method specifies a static type for its return value and all its arguments. An untyped method leaves all return and argument types unspecified. A method that is neither typed nor untyped as we has defined is classified as partially typed (*e.g.* A method that specifies a static type for all of its arguments but leaves its return type unspecified). Table 4.4 shows the numbers of elements in each category, for the different versions of the GZip and SAX libraries.

More precisely, for GZip we profiled the execution of the benchmark and typed the two methods whose interaction was causing most overhead: methods nextPutAll: and next:putAll:startingAt: of the DeflateStream class. For SAX-T1, we typed the interface between the driver SAXDriver and the handler SAXHandler. In SAX-T2, we also typed the interface between the driver and the tokenizer XMLTokenizer. We decided to type the interface between those classes because they are part of the core of a SAX Parser. Because of our expertise with this kind of parser, it was relatively easy to type those methods.

Every macrobenchmark is repeated thirty times and the average time is calculated. The benchmarks were run on a machine with an Intel Core i7 3.20 GHz CPU, 4 GB RAM and 250 GB SSD disk, running Ubuntu 12.10.

	Base ST (sec)	Exec (sec)	Hybrid (sec)	Hybrid vs Exec (%)
GZip	1.774	1.776	1.804	+1.58
GZip-T	–	1.804	1.790	-0.80
SAX	7.681	9.003	9.075	+0.80
SAX-T1	–	14.468	13.184	-8.87
SAX-T2	–	35.667	22.528	-36.84

Table 4.5: Running times of the macrobenchmarks. Max relative error: ±0.75%

4.6.2 Results

Table 4.5 shows the results of the macrobenchmarks. The maximum measured relative error in the macrobenchmarks is ±0.75%, with a confidence level of 95%. For the GZip benchmark, we see that both strategies perform similarly in all scenarios. With the untyped library, the hybrid strategy is slightly slower (2.98%), and it is marginally faster with a version of the library with few type annotations (0.80%). These factors are however negligible, and hence we did not pursue more advanced typing for this scenario. The reason why GZip is fairly stable irrespective of typing is that much of the time is spent in I/O and other primitives, which are out of reach of the type system.

In the case of the untyped SAX benchmark, both approaches also exhibit the same performance. Interestingly, in this case, as more type annotations are added to the SAX library, the hybrid strategy becomes noticeably more competitive: it is nearly 9% faster with SAX-T1, and almost 37% faster with SAX-T2.

Overall, these results are consistent with the microbenchmarks: available type information allows the hybrid strategy to use unguarded methods, which are faster than the cast-first methods used by the execution strategy.

Finally, we see that as type annotations are added to a library, the performance tends to degrade, irrespective of the chosen cast insertion strategy. While this phenomenon is not perceptible in the case of GZip, it is substantial for SAX. The reason of the degradation is that, by adding type information, we create boundaries with dynamically-typed sections of the program. At these boundaries, casts have to be inserted to ensure that the static type assumptions are not violated when this code is called from the dynamically typed world. Since casts are performed eagerly, the entailed verification is costly compared to the default behavior of

Smalltalk, which does not perform eager verification and only raises an exception when a method is not found. Reflecting on this result and the results of the microbenchmarks for the hybrid strategy (discussed in Section 4.5.2) suggests that predicting the performance impact of adding static type information is not trivial. Using only the results of the microbenchmark, we could assume that typing more code in a language with a gradual type system would always improve performance. However, this is generally not so. It remains to be studied whether there is a tipping point beyond which adding more type annotations enables better absolute performance, or at least does not degrade it. Unfortunately, because of the lack of gradual typed programs or programmers practices with gradual typing, it is not possible to validate the usefulness of such a study. If we stick to a relative comparison between the execution and hybrid strategies, the results confirm that hybrid is progressively more advantageous as more type annotations are added.

4.7 Comparing Strategies beyond Performance

Until now we have focused on the performance of cast insertion strategies. In this section we discuss the impact of these strategies on memory consumption, modularity, and the interaction with inheritance.

4.7.1 Memory

As stated in Section 4.5.1, the hybrid strategy trades memory for speed: for each method in the source program, it generates both an unguarded version of the method, whose body is the same as the original method, and a guarded version, which additionally performs argument casts. Table 4.6 shows the memory usage of the Smalltalk image—which includes 7,314 classes and 67,066 methods—compiled with all three strategies, compared to the memory usage of the standard image. Note that we also report on an alternative implementation of the hybrid strategy (Hybrid-fwd), discussed below.

The results confirm that the hybrid strategy uses substantially more memory than the other strategies. This is unsurprising: the high memory overhead comes from the duplication of all methods. The overhead of the call strategy, on the

Strategy	Size (MB)	Overhead vs. Base ST
Call	6.08	+56.8%
Execution	4.68	+21.3%
Hybrid	6.81	+75.9%
Hybrid-fwd	5.93	+53.9%
Base ST	3.88	

Table 4.6: Memory footprint of Smalltalk images compiled using the different strategies.

other hand, is entirely due to call site transformations (casting arguments and type information retrieval at runtime), which turn out to be quite space consuming.

Different implementations of the hybrid strategy are possible, each with a different tradeoff between performance and memory consumption. First, instead of duplicating all methods, it is possible to make each guarded method call the corresponding unguarded method. This approach, called Hybrid-fwd in Table 4.6, avoids duplicating method bodies and clearly reduces the overhead down to +53.9%, which is slightly better than the memory overhead of the call strategy. The relatively high memory overhead of this particular implementation of the hybrid strategy compared to the execution strategy (around 30% more when compared to the baseline) can be explained by the fact that many methods are small. For a small method, introducing an extra forwarder method (which is also small) is as consuming as duplicating it.

Of course, the Hybrid-fwd approach saves some space at the expense of an extra method call in each guarded method. Because Smalltalk does not support statically-bound private methods, the added calls turn out to have a noticeable overhead on the SAX macrobenchmarks (Table 4.7). In any case, the results show that it is a viable alternative if memory consumption becomes an issue, for example, when programming for an embedded system. We conjecture that this overhead would be negligible in a language like Java where private method calls can be aggressively optimized.

Other implementations can also be considered. For example, an implementation could use only one method and pass an additional boolean parameter that determines whether casts should be performed or not. This would be fairly efficient space-wise, but comes at the cost of passing an additional argument and adding a

	Hybrid (sec)	Hybrid-fwd (sec)	H-fwd vs H (%)
GZip	1.804	1.839	+1.94
GZip-T	1.790	1.824	+1.90
SAX	9.075	10.980	+20.99
SAX-T1	13.184	14.174	+7.51
SAX-T2	22.528	25.423	+12.85

Table 4.7: Macrobenchmark running times of the duplication-based vs. forward-based implementations of the hybrid strategy.

branch in the code. The optimal version would be to allow a single method to have two entry points: one entry at the start of the method where the arguments are cast, and another entry just after these casts. The translation would then insert calls to the second entry point when casts can be safely skipped. This would however require support at the level of the virtual machine. We have not fully explored these different implementations so far.

4.7.2 Modularity

Suppose that we modify MyCollection (Listing 4.1) so that the type of elements stored changes from Integer to Date, however we do not recompile Client. In the case of the untyped client, all of the strategies would detect the mismatch and their casts would fail. However, in the case of the typed client, only the casts in the execution strategy would fail.

The reason for this is that the call strategy relies on the possibility to analyze all call sites of a given method in order to introduce the casts of arguments, if needed. If Client is not recompiled, then its implementation still assumes, wrongly, that the argument to addElement: has to be of type Integer. Conversely, the execution strategy casts arguments in the callee, and therefore does not need to re-check callers after such a change. Gradualtalk addresses the need for analyzing all the callers of a method when using the call strategy through a dependency tracking mechanism (Section 3.6). It triggers recompilation of all call sites of a given method when needed, causing required casts to be inserted accordingly.

To be able to introduce efficient calls to unguarded methods, the hybrid strategy also needs to analyze all callers of a given method and may need to introduce casts in the callers. This being said, if the choice is to favor modular recompilation instead of performance, it would be possible to configure the hybrid strategy so that certain modules are not transformed, therefore calling guarded methods.

Note that the dependency between callers and callees that manifests itself when using typed clients with the call or hybrid strategies is the same as the dependencies between typed components in any typed language[1]. Put succinctly: when shared assumptions are changed, both parties need to be rechecked. In contrast, the execution strategy is inherently modular (though less performant) in such scenarios, performing such rechecking dynamically.

4.7.3 Interaction with inheritance

The calculus of gradually-typed objects of Siek and Taha does not include any form of inheritance, and therefore issues related to overriding are not considered. In their work on gradual typing for first-class classes, Takikawa *et al.* observe that a standard subtyping approach "would fail because a subclass may override a method with a different type" [55]. Consequently, they use row polymorphism instead of standard subtype polymorphism.

More precisely, if overriding a method is valid whenever the overriding method is a *consistent* subtype of the overriden method, the call cast insertion strategy is unsound. Consider the following example:

```
A ≫ m: x          "superclass, untyped"

B ≫ m: (Integer) x    "subclass overrides with typed argument"
```

and the following client:

```
|(A) a|
a = B new.
a m: 'hi'
```

The static type of a is A and the signature of m: in A does not specify any argument type. Therefore the call strategy accepts the invocation of m: without inserting any cast to Integer. Hence the use of the call strategy would result in an unsound execution as the body of B.m executes with an argument of an invalid type.

[1]This is what Gilad Bracha calls the "anti-modularity" of types: http://gbracha.blogspot.com/2011/06/types-are-anti-modular.html

A possibility to retain soundness is to simply restrict valid overridings to proper subtyping, and not consistent subtyping. This however means that typed and untyped hierarchies cannot be mixed: a typed method can never be overridden by a untyped method, and vice versa. Remarkably, the execution strategy does not suffer from this soundness issue at all. This is because casts are inserted in the callees, the first thing B.m does is to cast its argument to Integer, which fails as expected. Therefore it is possible to define valid overriding based on consistent subtyping, but at the cost of sacrificing the efficiency benefits of the call strategy.

Again, the hybrid strategy provides the opportunity to achieve the best of both worlds: retaining soundness while exploiting opportunities for optimizations. To properly deal with the case above, the hybrid strategy needs to refrain from using the unguarded method call in case the expected type of an argument is Dyn. Nonetheless, it may still use the unguarded method when it is safe to do so. Consider the following client:

```
|(B) b|
b = B new.
b m: 1
```

The invocation of m: can be performed efficiently without any cast, because the static type of b specifies that the argument must be an Integer, which it is. In order to deal with the dual case of overriding, *i.e.* , a typed method is overriden by a dynamically-typed one, the guarded method must perform specific checks to ensure that the dynamic method is used only in ways that are compatible with the subtyping relation. More precisely, the arguments to the dynamic method should be either supertypes or subtypes of the declared argument types in the overriden method (supertypes are valid because of the contravariance in argument types)[1]. Also, the value returned by the dynamic method should be a subtype of the declared return type. Consider the following:

```
C ≫ m: x        "subclass of B, untyped"
```

Then the untyped client code:

```
c = C new.
c m: 'hi'
```

[1]This corresponds to the valid assignment relationship $\Longleftrightarrow$ in Dart [18, §15.4].

This invocation raises a cast error at runtime because C.m is used in a way that is incompatible with any typed overriding of B.m: subtyping-wise, String is unrelated to Integer. We leave the detailed, formal treatment of this approach for future work.

4.8 Conclusion

This chapter studies different cast insertion strategies for a gradually-typed language with objects. Experiments are carried out in Gradualtalk, a gradually-typed Smalltalk, and focus mainly on performance. Starting from the direct implementation of the semantics specified by Siek and Taha [51], which we refer to as the call strategy, and present two alternative strategies: one that inserts casts at the callee side, termed the execution strategy, and a hybrid strategy that combines ideas of the call and execution strategies. The execution and hybrid strategies build upon the fact that the language runtime is already safe, and therefore we can discharge casts from their safety-bearing role as explained in [51].

Microbenchmarks exhibit the best and worst cases of the call and execution strategy, and show that the hybrid strategy is effectively a best-of-both-worlds approach, always exhibiting a performance similar to that of the fastest strategy. A set of macrobenchmarks help us to further characterize the benefits of the hybrid strategy, which manifest themselves more clearly as more type annotations as added.

We also compare the three strategies on different qualities instead of only execution time. We report on the extra memory cost of two different versions of the hybrid strategy which we believe are reasonable in view of the associated benefits. Considering modular compilation, the execution strategy is better, although the hybrid strategy allows fine-tuning of the modularity/efficiency tradeoff. Finally, we discuss the interaction of these strategies with inheritance: the call strategy is unsound, the execution strategy is sound but sacrifices performance. We informally describe a way to adapt the hybrid strategy to retain soundness and still exploit static type information for optimization.

Overall, this suggests that the hybrid strategy is a promising approach to implement gradual typing in an existing dynamic language with a safe runtime.

Chapter 5

Confined Gradual Typing

In the previous chapter, we presented three different cast insertion strategies and demonstrated how they can affect performance. In this chapter[1], we present Confined Gradual Typing, an extension to Gradual typing that refines it with annotations that allow programmers to explicitly prohibit certain boundary crossings. We present the formal description, implementation and validation of Confined Gradual Typing for two variants: Strict Confined Gradual Typing and Relaxed Confined Gradual Typing.

5.1 Introduction

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 carrying out 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,

[1]This chapter is based on the publication: 'Confined Gradual Typing" [3].

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 an 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 predictability. 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 (Chapter 3) as a practical testbed.

This chapter is structured as follows: Section 5.2 illustrates the issues associated with the implicitness of gradual typing, and informally describes Confined Gradual Typing in its two variants. Sections 5.3 and 5.4 formalize Strict Confined Gradual Typing and Relaxed Confined Gradual Typing respectively, establishing the key properties of each approach. Section 5.5 briefly describes the implementation of both variants in Gradualtalk. Section 5.6 reports on performance measurements of CGT in Gradualtalk, highlighting the incurred cost of higher-order casts. Section 5.7 reviews related work, and Section 5.8 concludes.

All the proofs associated with the formalization are included in Appendix A. The implementation and benchmark code are available online.[1]

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

5.2 Motivation

In this section we 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.

5.2.1 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 (see Chapter 3). 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
```

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 drop-down list 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.[1] 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 offer such a guarantee, because all values can implicitly cross to the untyped world.

[1]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 [22, 58, 39, 62] addresses *traceability*—by reporting the source of a runtime cast error—not *reliability*, the absence of runtime cast errors.

5.2.2 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 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 explained in Section 2.1, a function wrapper is a function of the expected type that internally inserts casts to the arguments and the returned value of the underlying function. 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 5.6.1, 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 2.2.4). We were faced with exactly this problem when performing benchmarks as part of the validation of our work on cast insertion strategies (Chapter 4), and wished we had a way to predict and control where wrappers are introduced.

5.2.3 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 chapter 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.

5.2.3.1 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 5.2.1).[1]

For instance, if the statistics block from the UI team is typed as ↑(Integer Integer → 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 ↑. 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 ↓ 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 5.2.2). For instance, the developer of the typed collection library can provide a second fold operation inject:intoSafe: where the argument block is typed with ↓:

```
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 5.2.2 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.

5.2.3.2 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.

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

In RCGT, the $\uparrow$ qualifier, as in $\uparrow T$, expresses the constraint that the (higher-order) value of type T will not be wrapped. The $\downarrow$ 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 $\downarrow$(Integer Integer $\rightarrow$ 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 $\rightarrow$ Dyn, and the following program, statically legal in both SCGT and RCGT:

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

If we protect f with the $\uparrow$ 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 error 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.

5.2.4 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.

5.3 Strict Confined Gradual Typing

To make the description of Confined Gradual Typing precise, we now formalize CGT, starting with the strict variant (Section 5.2.3.1). We defer the presentation of the relaxed variant (Section 5.2.3.2) to Section 5.4.

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 [50], 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.[1]

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 5.3.1. Section 5.3.2 presents the internal language, and the translation is explained in Section 5.3.3.

5.3.1 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 5.1).

Subtyping. The $\uparrow$ and $\downarrow$ qualifiers induce a natural subtyping relation between types (Figure 5.2). Since $\downarrow$ is a guarantee about the past of a value, it is possible

[1]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 5.5.

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

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 5.1: SCGT: Source language syntax

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

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

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

Figure 5.2: SCGT: Static subtyping

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 the qualifiers (SS-up, SS-down). The other rules are standard.

Directed consistency. As Section 2.2.1 explained, the essence of gradual typing lies in the *consistency* relation [50], which expresses the compatibility between typed and untyped expressions. 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 5.3). 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

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

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

$$\text{(DC-fun)}\frac{T_3 \rightsquigarrow T_1 \qquad T_2 \rightsquigarrow T_4}{T_1 \to T_2 \rightsquigarrow T_3 \to T_4} \qquad \text{(DC-losedown)}\frac{T_1 \rightsquigarrow T_2}{\downarrow T_1 \rightsquigarrow T_2}$$

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

Figure 5.3: SCGT: Directed consistency

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.

Typing. Equipped with directed consistency, we can now describe the typing rules of SCGT (Figure 5.4). 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 [50], 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:

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

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

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

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

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

Figure 5.4: SCGT: Typing

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 $\downarrow$.

5.3.2 Internal Language

Syntax. Figure 5.7 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 $T \neq \text{Dyn}$). 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.

$$\text{(IT-num)}\ \frac{}{\Gamma \vdash n : \downarrow\text{Int}} \qquad \text{(IT-var)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

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

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

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

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

Figure 5.5: SCGT: Internal language typing

Typing. The typing rules of the internal language are straightforward (Figure 5.5). 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 5.3.3). 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). The evaluation rule (E-fcastinv) describes the application of a function wrapper to a value. Regardless of the qualifiers of F and F', it decomposes to cast the argument and result of the application.

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

$$\text{(E-add)}\ \frac{n_3 = n_1 + n_2}{n_1 + n_2 \longrightarrow n_3}$$

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

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

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

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

Figure 5.6: SCGT: Internal language dynamic semantics

5.3.3 Translating Source Programs to the Internal Language

A source language program is translated to an internal language program through cast insertion (Figure 5.8). Each typing rule of the source language (Figure 5.4) 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

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

Figure 5.7: SCGT: Internal language syntax

expression (C-app2). (C-add) is similar.

5.3.4 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 (Appendix A.1). □

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 (Appendix A.1). □

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 (Appendix A.2). □

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.

$$\text{(C-num)}\frac{}{\Gamma \vdash n \Rightarrow n : \downarrow\mathrm{Int}} \qquad \text{(C-var)}\frac{\Gamma(x) = T}{\Gamma \vdash x \Rightarrow x : T}$$

$$\text{(C-abs)}\frac{\Gamma, x : T_1 \vdash e \Rightarrow e' : T_2}{\Gamma \vdash \lambda x : T_1.e \Rightarrow \lambda x : T_1.e' : \downarrow(T_1 \to T_2)}$$

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

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

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

Figure 5.8: SCGT: Cast insertion

The proof technique we use consists of formulating a variant of the semantics of SCGT where values are marked.[1] 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.3.

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 $\emptyset \vdash \langle Dyn \Leftarrow T \rangle\ v : Dyn$, then $v \neq \widehat{v}$.*

[1]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 [28].

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.3), 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.3). □

Finally, we establish that the taint-tracking semantics is faithful to the semantics presented in this section by defining a taint erasure function $erase(e)$ 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* $erase(e) \longrightarrow erase(e')$.

Proof. By induction on the evaluation rules of the taint-tracking semantics (Appendix A.3). □

5.4 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 unwanted and costly function wrappers are not created. 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 5.4.1). Its semantics is given by translation (unchanged) to an internal language with casts, for which only one evaluation rule is different (Section 5.4.2). The metatheory is fairly different (Section 5.4.3) 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 5.1): $\uparrow P = \downarrow P = P$

5.4.1 Directed Consistency, Revisited

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

Figure 5.9: 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 5.9): 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.

5.4.2 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 5.10. 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.

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

Figure 5.10: 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 where wrappers do not get created, instead raising a runtime exception.

For instance consider a typed function $f : F$, the dynamically-typed identity function, $id = \lambda x.x : \mathrm{Dyn} \rightarrow \mathrm{Dyn}$, 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 $\uparrow F \not\leadsto \text{Dyn}$. 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, $\uparrow F \not\leadsto F'$—the $\uparrow$ qualifier cannot be lost. So, (E-merge-err) applies, resulting in a CastError that states that the wrapper creation is illegal.

5.4.3 Type Safety and Correctness of Qualifiers

Progress and preservation also hold for Relaxed Confined Gradual Typing and similarly, cast insertion preserves typing (Appendix A.4). 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 $e = \langle F_2 \Leftarrow F_1\rangle\ f$*,* e *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: $T_1 \leadsto \downarrow T_2 \Rightarrow T_1 <: \downarrow T_2$ and $\uparrow T_1 \leadsto T_2 \Rightarrow \uparrow T_1 <: T_2$. Then, the proof proceeds by contradiction, exploiting the definition of a wrapper value, *i.e.* $T_1 \not<: T_2$ (Appendix A.5). □

5.5 Implementation

We have implemented both variants of Confined Gradual Typing, SCGT and RCGT, as an extension of Gradualtalk (Chapter 3), a gradually-typed dialect of Smalltalk. The implementation can be configured to operate in one of three modes: original gradual typing (hereinafter GT), SCGT and RCGT. This section briefly describes how to go from the theory of Confined Gradual Typing to its implementation in Gradualtalk.

5.5.1 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 (Chapter 3) is an object-oriented language, with subtyping. Gradualtalk is built on *consistent subtyping* [51], a relation that combines consistency with traditional object subtyping. We first extend subtyping in Gradualtalk to include the subtyping rules for qualifiers (Figure 5.2). Second, we make the consistent subtyping relation *directed* (Figure 5.3). 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, 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 5.5.2.

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 (Section 3.6), and CGT does not introduce new challenges in this regard.

5.5.2 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 5.2) 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.[1]

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 5.6 and 5.10):

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:

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

   (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.

5.6 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 micro and macrobenchmarks. For all practical purposes, Strict Confined Gradual Typing has the same performance as an original gradually typed system. This is because the difference between the two is in the typechecker which runs at compile time, while the runtime is the same for both. For this reason, it is ommited in the benchmarks.

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 also check to 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.

5.6.1 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 5.2. 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 5.2.1, with type Integer Integer → 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 B).

5.6.1.1 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. The benchmark loop is then 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 5.1 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 5.5.2). The Wrapping variant takes about 13 seconds in GT and about 17 seconds in RCGT. This means that wrapper 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 5.11). 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.

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

Table 5.1: Execution time in milliseconds for one millon boundary crossings back into typed code. RCGT is about 30% slower than GT.

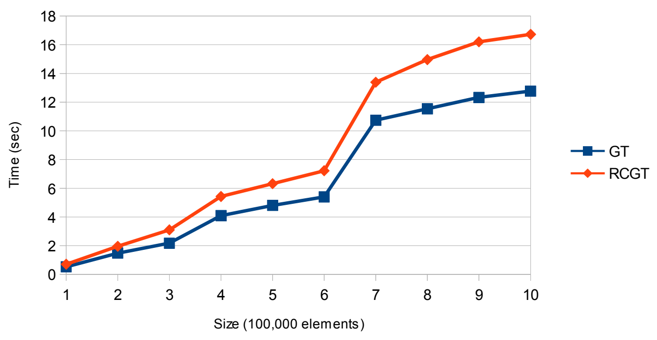

Figure 5.11: Running times for creating wrappers in GT and RCGT. RCGT is systematically about 30% slower.

5.6.1.2 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	Wrapping
GT	2246	25458
RCGT	2648	26359
RCGT with $\downarrow$	2694	-
RCGT with $\uparrow$	2660	-

Table 5.2: Execution time in milliseconds for one million applications of the closure. Wrapper evaluation implies about a 10X slowdown. The maximum measured relative error in the microbenchmarks is ±13.48%, with a confidence level of 95%.

- **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 and col 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 5.2 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 5.12). The reason of this overhead is that casting the arguments and return is a very expensive and complex action compared with the actual code to execute. Adding two numbers is faster compared to checking the type of the two arguments, no matter the quality of interpreter or JIT compiler. Also, the use of type qualifiers in the No Wrapping variant does not affect the performance of RCGT.

5.6.2 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 [46]. 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. Unfortunately, we could not reuse the macrobenchmarks used in Cast Insertion Strategies (Chapter 4) because these applications does not use blocks in their operation.

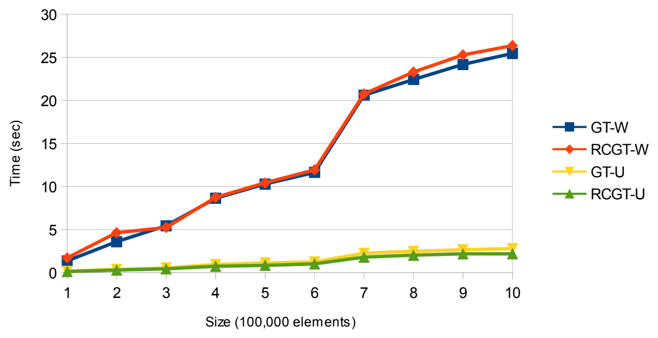

Figure 5.12: Running times for closure application when not wrapped (GT-U, RCGT-U) and wrapped (GT-W, RCGT-W). Wrappers always implies about a 10X slowdown.

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 providing 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, instead of, for example, 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 3,620 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 [1].

Results The macrobenchmark has four different variants: GT and RCGT, both with and without wrappers. The detailed results are in Table 5.3. When the block is wrapped, GT and RCGT take about 1.6 seconds, compared to 1.1 seconds when the

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

	Not Wrapped	Wrapped
GT	1113	1573
RCGT	1123	1608

Table 5.3: Execution time in milliseconds for the macrobenchmark. Wrapping induces a 45% slowdown.

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.

5.6.3 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.

5.7 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 [57] 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 are. As such, we believe that this technique is perfectly compatible with module-level approaches, and can be used to extend their expressive power.

Rastogi et al. [44], 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. [30] 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 [52]. These space concerns are orthogonal to the value-flow concerns addressed by Gradual Gradual Typing, and we believe that the two can be easily integrated.

Recently, Swamy et al. [54] 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 [39, 58, 62, 49]. 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.

5.8 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.

Chapter 6

Conclusion

The goal of this thesis is to improve the performance and reliability of gradually typed applications. Our work reduced the performance hit of using gradual typing by improving the cast insertion process and by performing removal of runtime checks in specific cases. It also provided a solution to the reliability issue by allowing programmers to specify using types which values cannot come from or go to dynamically typed code. Although there is still a lot of work to do to solve both problems, our work makes significant steps towards addressing them.

In this chapter, we present a summary of our contributions and provide perspectives on what are the next steps to follow.

6.1 Summary of the work

In this work, we make three contributions: Gradualtalk, Cast Insertion Strategies and Confined Gradual Typing. Each of these contributions have resulted in the following publications:

- **Gradual Typing for Smalltalk** [2]: This journal paper describes Gradualtalk and its features, as well as a preliminary empirical validation of its design (not included in this thesis).

- **Cast Insertion Strategies for Gradually-Typed Objects** [4]: This conference paper describes the different cast insertion strategies and validates them.

- **Confined Gradual Typing** [3]: This conference paper presents Confined Gradual Typing (CGT) in two variants. The paper provides a formal description and proof for each variant. Finally, it presents an implementation for CGT and shows that there is no significant performance hit for using CGT.

In the following subsections, we provide a summary of each contribution.

6.1.1 Gradualtalk

Gradualtalk is a gradually typed Smalltalk. The type system of Gradualtalk combines several state-of-the-art features, such as gradual typing, unified nominal and structural subtyping, self type constructors for metaclasses, and blame tracking. Gradualtalk is designed to ease the migration of existing, untyped Smalltalk to typed Gradualtalk code. Gradualtalk was also developed to be the foundation of our research about improving the performance of gradually typed applications.

The development of Gradualtalk and the typing of the corpus for the validation provided valuable information of the problems that are addressed in both Cast Insertion Strategies and Confined Gradual Typing. These problems are performance degradation and reliability of being able to ensure that a statically typed code will never throw a runtime type error.

6.1.2 Cast Insertion Strategies

When using a gradual type system, applications incur a performance degradation. One of the reasons for this is the casts inserted to provide the guarantees of gradual typing, especially the casts inserted in method invocations. We studied two different cast insertion strategies for a gradually-typed language with objects: call and execution. The call strategy inserts casts at call sites whenever needed. The execution strategy inserts casts on arguments of a statically-typed method directly at the beginning of the method. The call strategy is efficient for statically typed code, but inefficient for dynamically typed code. In contrast, the execution strategy is efficient for dynamically typed code, but inefficient for statically typed code.

We proposed the hybrid strategy, which merges the best of both strategies. This is done by duplicating each method: one version is the original method code that does not cast its arguments, called the *unguarded* method, and the other method starts by casting its arguments, called the *guarded* method. Then, the strategy has two different scenarios for inserting casts. For statically-typed receivers, the compiler modifies the call site to invoke the unguarded method. For dynamically-typed receivers, the compiler leaves the call site intact, thereby calling the guarded method.

We validated our claims about the performance of the call, execution and hybrid strategy with both micro and macro benchmarks. The microbenchmarks exhibit the best and worst cases of the call and execution strategy, and show that the hybrid strategy is effectively a best-of-both-worlds approach. A set of macrobenchmarks help us to further characterize the benefits of the hybrid strategy, which manifest themselves more clearly as more type annotations are added. The hybrid strategy goes from a 9% speedup to almost 37% speedup compared with the execution strategy when more type annotations are added.

6.1.3 Confined Gradual Typing

There are at least two problems with gradual typing: reliability and performance. The reliability problem corresponds to ensuring that statically typed code will not throw a runtime type error. The performance problem is that execution of wrapped closures is costly, and can be silently incurred in statically-typed code due to the flow of higher-order values. These two problems can be addressed with Confined Gradual Typing. Confined Gradual Typing refines a gradual type system with annotations to allow programmers to explicitly prohibit certain boundary crossings between statically typed code and dynamically typed code. Two type qualifiers are introduced: $\uparrow$ and $\downarrow$. $\uparrow$ 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.

There are two variants: Strict Confined Gradual Typing and Relaxed Confined Gradual Typing. In Strict Confined Gradual Typing, the qualifiers express that an expression, once reduced to a value, has never flowed from or cannot flow into the untyped world, depending of the qualifier. In contrast, in Relaxed Confined Gradual Typing, all values can flow from or into the untyped world. When closures flow from or into the untyped world, they can be wrapped to check at runtime that their arguments passed and the return value have the correct type. However, these wrappers have a cost in performance. In Relaxed Confined Gradual Typing, the qualifiers express that the reduced value has never been wrapped or it cannot be wrapped.

We formally described both variants of Confined Gradual Typing and proved that these variants are type sound and the qualifiers actually provide the guarantees we expect. We also showed using both micro and macrobenchmarks that the performance cost of implementing Relaxed Confined Gradual Typing is negligible compared to the cost of wrapping.

6.2 Perspectives

6.2.1 Extensions

Confined Gradual Typing could be extended in two ways: mixing variants and multiple kinds of dynamically typed code.

Mixing variants We presented two variants for Confined Gradual Typing: Strict and Relaxed. We developed these two variants separately. However, we can see the potentiality of having these two variants together. For example, one library wants to prevent the wrapping of the objects it creates while another library wants to enforce that all values passed to itself are statically typed. To guarantee that this combination works as expected, there are two requirements needed: a formal definition and formal proofs of type soundness and qualifier correctness.

Multiple kinds of Dyn The work of Swamy *et al.* [54] showed that there can be different kinds of dynamically typed code. In the case of the work of Swamy *et al.*, one kind of dynamically typed code was a checked one, still under control of the typechecker, while the other kind was unchecked, never checked by the typechecker. There are three questions that arise when trying to implement Confined Gradual Typing when having more than one kind of dynamically typed code:

1. How can Confined Gradual Typing be defined when there are two kinds of dynamically typed code?
2. Can it be useful to have more than two kinds of dynamically typed code? What could a third category represent?
3. Who makes the decision of how many kinds of dynamically typed code there are: The language designer or the application programmer?

Answering these three questions and exploring the ramifications of the answers would provide a solution to the integration between Confined Gradual Typing and multiple kinds of Dyn.

6.2.2 Formalization

We have proven that Confined Gradual Typing in its two variants is type sound and the type qualifiers we presented actually provide the guarantees we expect. In contrast, both the execution strategy and hybrid strategy presented as alternative strategies to perform cast insertion lack a formal proof. To give this formal proof, two steps are required. The first step is to formally describe the transformation that each strategy realizes. For this, both a source and target formal language where the transformation operates is required. The second step is to prove that the transformation preserves the typing between the source and target expression.

6.2.3 Inheritance and Modularity

When describing the hybrid strategy, we mentioned how we can use this strategy to provide standard subtyping polymorphism in a type sound manner. However, this description is informal and lacks a proof that it is type sound. The description was left informal because we have not rigorously looked at the interaction between statically typed and dynamically typed code in presence of inheritance. This is an open challenge for future work.

6.2.4 Case studies

We used both micro and macro benchmarks to validate our affirmations about the cast insertion strategies and Confined Gradual Typing. However, in both cases the benchmarks were designed by ourselves. This was because the Smalltalk community lacks a standard set of benchmark suites. Hence, these benchmarks can still be biased. Also, our benchmarks, in scope were relatively small and limited, compared with standard benchmarks in the industry. To further validate our affirmations, a different set of micro and macro benchmarks can be used.

Another issue with our case studies is that we were focused on performance. Another area that still needs further validation is the impact of Confined Gradual Typing on programmers. We need to study how Confined Gradual Typing is used. For that, it is necessary to first realize empirical studies about gradual typing. After that, it is needed to realize empirical studies that compare the usage between Gradual Typing and Confined Gradual Typing.

Appendix A

Confined Gradual Typing: Formal Proof

A.1 SCGT: Type Safety

Lemma 1. *(Inversion)*

1. $\emptyset \vdash n : T$, *then* $|T| = Int$
2. $\emptyset \vdash \lambda x : T'.e : T$, *then* $|T| = T_1 \rightarrow T_2$
3. $\emptyset \vdash \langle F \Leftarrow F' \rangle f : T$, *then* $|T| = T_1 \rightarrow T_2$
4. $\emptyset \vdash \langle Dyn \Leftarrow T \rangle b : T$, *then* $T = Dyn$

Proof. By case analysis:

Case v=n

1. $\emptyset \vdash n : \downarrow\text{Int}$, by (IT-num)
2. $\emptyset \vdash n : T$, $|T| = \text{Int}$

Case v=$\lambda x : T.e$

1. $\emptyset \vdash \lambda x : T.e : \downarrow(T \rightarrow T')$, by (IT-abs)
2. $\emptyset \vdash \lambda x : T.e : T''$, $|T''| = T \rightarrow T'$

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

1. $\emptyset \vdash \langle F \Leftarrow F' \rangle f : F$, by (IT-cast)
2. $\emptyset \vdash \langle F \Leftarrow F' \rangle f : F$, $|F| = T \rightarrow T'$ by definition of F

Case v=$\langle$Dyn $\Leftarrow T\rangle b$

1. $\emptyset \vdash \langle \text{Dyn} \Leftarrow T\rangle b : \text{Dyn}$, by (IT-cast)

□

Lemma 2. *(Canonical forms)*

1. $\emptyset \vdash v : T$ *and* $|T| = Int$*, then* $\exists n,\ v = n$
2. $\emptyset \vdash v : T$ *and* $|T| = T_1 \to T_2$*, then* $v = f$
3. $\emptyset \vdash v : T$ *and* $T = Dyn$*, then* $v = \langle Dyn \Leftarrow T\rangle b$

Proof. By case analysis:

Case $|T| =$ Int

1. $v = n$, by Lemma 1-1

Case $|T| = T_1 \to T_2$

1. $v = \lambda x : T.e$ or $v = \langle F \Leftarrow F'\rangle f$, by Lemma 1-2,3
2. $v = f$

Case $T =$ Dyn

1. $v = \langle \text{Dyn} \Leftarrow T\rangle b$, by Lemma 1-4

□

Theorem 8. *(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 type rules.

Case (IT-num), (IT-abs)

1. e is a value

Case (IT-err)

1. e is CastError

Case (IT-var)

1. Case impossible with an empty environment (Well-typed)

Case (IT-app)

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

Case (IT-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 (E-congr) with $E = \Box e_2$ to progress.
3. If e_1 is CastError, use rule (E-err) with $E = \Box e_2$ to progress.
4. If e_1 is a value and e_2 is not a value, use rule (E-congr) with $E = e_1 + \Box$ to progress.
5. If e_1 is a value and e_2 is CastError, use rule (E-err) with $E = e_1 + \Box$ to progress.
6. If e_1' and e_2 are both values, then $e_1 = n_1$ and $e_2 = n_2$ by Lemma 2-1. Use (E-add) to progress.

Case (IT-cast)

1. By assumption:
   (a) $\emptyset \vdash \langle T_3 \Leftarrow T_2 \rangle e : T_3$
   (b) $\emptyset \vdash e : T_1$, $T_1 <: T_2$, $T_2 \rightsquigarrow T_3$
2. If e is not a value, use rule (E-congr) with $E = \langle T_3 \Leftarrow T_2 \rangle\ \Box$ to progress.
3. If e is CastError, use rule (E-err) with $E = \langle T_3 \Leftarrow T_2 \rangle\ \Box$ to progress.
4. If e is a value and $T_2 <: T_3$, use rule (E-remove) to progress.
5. If e is a value and $T_2 \rightsquigarrow T_3$ and $T_2 \not<: T_3$, then $\exists F_2, F_3\ T_2 = F_2$ and $T_3 = F_3$. Then $\langle T_3 \Leftarrow T_2 \rangle e$ is a value

$\Box$

Lemma 3. *(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 (IT-num)

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

Case (IT-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 (IT-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 (IT-abs)

1. By assumption:
   (a) $\Gamma, x : T \vdash \lambda y : T'.e : \downarrow(T' \rightarrow T'')$
   (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. $\Gamma \vdash \lambda y : T'.(e[v/x]) : \downarrow(T' \rightarrow T'')$, by (IT-abs)
5. $\Gamma \vdash (\lambda y : T'.e)[v/x] : \downarrow(T' \rightarrow T'')$, by Substitution definition

Case (IT-err)

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

Case (IT-app)

1. By assumption:
   (a) $\Gamma, x : T \vdash e_1\ e_2 : T_{12}$
   (b) $\Gamma, x : T \vdash e_1 : T_1$
   (c) $\Gamma, x : T \vdash e_2 : T_2$
   (d) $|T_1| = T_{11} \rightarrow T_{12}$, $T_2 <: T_{11}$
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 (IT-app)
5. $\Gamma \vdash (e_1\ e_2)[v/x] : T'$, by Substitution definition

Case (IT-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 (IT-add)
5. $\Gamma \vdash (e_1 + e_2)[v/x] : \downarrow\text{Int}$, by Substitution definition

Case (IT-cast)

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

□

Theorem 9. *(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.

Case (E-congr)

Subcase $E = \square\ e$

1. By assumption:
   (a) $e_1\ e_2 \longrightarrow e_1'\ e_2$
   (b) $\emptyset \vdash e_1\ e_2 : T_{12}$, $\emptyset \vdash e_1 : T_1$, $\emptyset \vdash e_2 : T_2$, $|T_1| = T_{11} \rightarrow T_{12}$, $T_2 <: T_{11}$
   (c) $e_1 \longrightarrow e_1'$
2. $\emptyset \vdash e_1' : T_1'$, $T_1' <: T_1$, $|T_1'| = T_{11}' \rightarrow T_{12}'$, 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 (IT-app) and (4)

Subcase $E = v\ \square$

1. By assumption:
   (a) $e_1\ e_2 \longrightarrow e_1\ e_2'$
   (b) $\emptyset \vdash e_1\ e_2 : T_{12}$, $\emptyset \vdash e_1 : T_1$, $\emptyset \vdash e_2 : T_2$, $|T_1| = T_{11} \rightarrow T_{12}$, $T_2 <: T_{11}$
   (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 (IT-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 IT-add)
3. $T_1' <: \uparrow\text{Int}$, by induction on (1-c) and (SS-trans)
4. $\emptyset \vdash e_1' + e_2 : T_3$, by using (IT-add)

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

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) $\emptyset \vdash T_2 \Leftarrow T_1 \rangle\ e : T_2$, $\emptyset \vdash e : T'$, $T' <: T$
   (c) $e \longrightarrow e'$
2. $\emptyset \vdash e' : T''$, $T'' <: T'$ by induction of (1-c)
3. $T'' <: T$, by (SS-trans)
4. $\emptyset \vdash \langle T_2 \Leftarrow T_1 \rangle\ e' : T_2$ (IT-cast)

Case (E-app)

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

Case (E-add)

1. By assumption:
   (a) $v_1 + v_2 \longrightarrow v_3$
   (b) $\emptyset \vdash v_1 + v_2 : T$, $v_3 = v_1 + v_2$
2. $T = \downarrow\mathrm{Int}$ (From IT-add)
3. $\emptyset \vdash v_3 : \downarrow\mathrm{Int}$ (From IT-num)
4. $\emptyset \vdash v_3 : T$, replacing by T

Case (E-merge)

1. By assumption:
   (a) $\langle T_2 \Leftarrow \mathrm{Dyn}\rangle\langle \mathrm{Dyn} \Leftarrow T_1 \rangle\ v \longrightarrow \langle T_2 \Leftarrow |T_1| \rangle\ v$
   (b) $T_1 \rightsquigarrow T_2$
2. $\emptyset \vdash \langle \mathrm{Dyn} \Leftarrow T_1 \rangle\ v : \mathrm{Dyn}$, reverse (IT-cast) in (1-a)
3. $\emptyset \vdash v : T_1'$, $T_1' <: T_1$ reverse (IT-cast) in (2)
4. $T_1 \neq \uparrow T_1'$, because $\uparrow T_1'' \not\rightsquigarrow \mathrm{Dyn}$
5. $T_2 \neq \downarrow T_2'$, because $\mathrm{Dyn} \not\rightsquigarrow \downarrow T_2'$
6. $T_1 = \downarrow T_1'$ or $T_1 = |T_1'|$
7. $T_2 = |T_2'|$ or $T_2 = \uparrow T_2'$
8. $|T_1| \rightsquigarrow T_2$, (SS-losedown) and/or (SS-gainup)
9. $T_1 <: |T_1|$, by (SS-losedown) or (SS-reflex)
10. $T_1' <: |T_1|$, by (SS-trans) and (3)
11. $\emptyset \vdash \langle T_2 \Leftarrow |T_1| \rangle\ v : T_2$ (IT-cast)

Case (E-remove)

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

Case (E-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. $\emptyset \vdash \langle F' \Leftarrow F \rangle \ f : F'$, (IT-cast)
3. $\emptyset \vdash f : F''$, $F'' <: F$, $|F''| = T_1'' \to T_2''$ inverse (IT-cast)
4. $T_1 <: T_1''$, $T_2'' <: T_2$, by using (SS-fun)
5. $T <: T_1'$, inverse (IT-app) of (1-c) using (2)
6. $\emptyset \vdash \langle T_1 \Leftarrow T_1' \rangle \ v : T_1$, (IT-cast)
7. $\emptyset \vdash f \ \langle T_1 \Leftarrow T_1' \rangle \ v : T_2''$, (IT-app)
8. $\emptyset \vdash \langle T_2' \Leftarrow T_2 \rangle (f \ (\langle T_1 \Leftarrow T_1' \rangle \ v)) : T_2'$, (IT-cast)

Case (E-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 (IT-err)

$\square$

A.2 SCGT: Cast Insertion

Lemma 4. *$\Gamma \vdash \langle\langle T \Leftarrow T' \rangle\rangle \ e : T''$ and $\Gamma \vdash e : T'$, then $T'' <: T$*

Proof. By case analysis

Case $T' <: T$

1. Assumptions:
   (a) $\Gamma \vdash e : T'$
   (b) $T' <: T$
2. $T' <: T$, by (1-b)

Case $T' \not<: T$

1. Assumptions:
   (a) $\Gamma \vdash \langle T \Leftarrow T' \rangle\ e : T''$
   (b) $\Gamma \vdash e : T'$
2. $\Gamma \vdash \langle T \Leftarrow T' \rangle\ e : T$, by (IT-cast)

□

Theorem 10. *(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 type rules.

Case (C-num)

1. By assumption:
   (a) $\Gamma \vdash n \Rightarrow n : \downarrow\text{Int}$
2. $\Gamma \vdash n : \downarrow\text{Int}$, by using (IT-num)

Case (C-var)

1. By assumption:
   (a) $\Gamma \vdash x \Rightarrow x : T$
   (b) $\Gamma(x) = T$
2. $\Gamma \vdash x : T$, by using (IT-var)

Case (C-abs)

1. By assumption:
   (a) $\Gamma \vdash \lambda x : T_1.e \Rightarrow \lambda x : T_1.e' : \downarrow(T_1 \rightarrow T_2)$
   (b) $\Gamma, x : T_1 \vdash e \Rightarrow e' : T_2$
2. $\Gamma, x : T_1 \vdash e' : T_2$, by induction of (1-b)
3. $\Gamma \vdash \lambda x : T_1.e' : \downarrow(T_1 \rightarrow T_2)$, by using (IT-abs)

Case (C-app1)

1. By assumption:
   (a) $\Gamma \vdash e_1\ e_2 \Rightarrow (\langle(T_2 \rightarrow \text{Dyn}) \Leftarrow \text{Dyn}\rangle e_1')e_2' : \text{Dyn}$
   (b) $\Gamma \vdash e_1 \Rightarrow e_1' : \text{Dyn}$
   (c) $\Gamma \vdash e_2 \Rightarrow e_2' : T_2$
   (d) $T_2 \leadsto \text{Dyn}$
2. $\text{Dyn} \rightarrow \text{Dyn} \leadsto T_2 \rightarrow \text{Dyn}$, by (DC-fun)
3. $\text{Dyn} \leadsto T_2 \rightarrow \text{Dyn}$, by (DC-dynfun2)
4. $\Gamma \vdash e_1' : \text{Dyn}$ and $\Gamma \vdash e_2' : T_2$, by induction of (1-b) and (1-c)
5. $\Gamma \vdash (\langle(T_2 \rightarrow \text{Dyn}) \Leftarrow \text{Dyn}\rangle e_1') : T_2 \rightarrow \text{Dyn}$, by (IT-cast)
6. $\Gamma \vdash (\langle(T_2 \rightarrow \text{Dyn}) \Leftarrow \text{Dyn}\rangle e_1')e_2' : \text{Dyn}$, by (IT-app)

Case (C-app2)

1. By assumption:
   (a) $\Gamma \vdash e_1\ e_2 \Rightarrow e_1'(\langle\langle T_{11} \Leftarrow T_2\rangle\rangle e_2') : T_{12}$
   (b) $\Gamma \vdash e_1 \Rightarrow e_1' : T_1$
   (c) $\Gamma \vdash e_2 \Rightarrow e_2' : T_2$
   (d) $|T_1| = T_{11} \rightarrow T_{12}$ and $T_2 \leadsto T_{11}$
2. $\Gamma \vdash e_1' : T_1$ and $\Gamma \vdash e_2' : T_2$, by induction of (1-b) and (1-c)
3. $\Gamma \vdash \langle\langle T_{11} \Leftarrow T_2\rangle\rangle e_2' : T'$ and $T' <: T_{11}$, by Lemma 4
4. $\Gamma \vdash e_1'(\langle\langle T_{11} \Leftarrow T_2\rangle\rangle e_2') : T_{12}$, by (IT-app)

Case (C-add)

1. By assumption:
   (a) $\Gamma \vdash e_1 + e_2 \Rightarrow (\langle\langle \uparrow\text{Int} \Leftarrow T_1\rangle\rangle e_1') + (\langle\langle \uparrow\text{Int} \Leftarrow T_2\rangle\rangle e_2') : \downarrow\text{Int}$
   (b) $\Gamma \vdash e_1 \Rightarrow e_1' : T_1$
   (c) $\Gamma \vdash e_2 \Rightarrow e_2' : T_2$
   (d) $T_1 \leadsto \uparrow\text{Int}$ and $T_2 \leadsto \uparrow\text{Int}$
2. $\Gamma \vdash e_1' : T_1$ and $\Gamma \vdash e_2' : T_2$, by induction of (1-b) and (1-c)
3. $\Gamma \vdash (\langle\langle \uparrow\text{Int} \Leftarrow T_1\rangle\rangle e_1') : T_1'$ and $T_1' <: \uparrow\text{Int}$, by Lemma 4
4. $\Gamma \vdash (\langle\langle \uparrow\text{Int} \Leftarrow T_2\rangle\rangle e_2') : T_2'$ and $T_2' <: \uparrow\text{Int}$, by Lemma 4
5. $\Gamma \vdash (\langle\langle \uparrow\text{Int} \Leftarrow T_1\rangle\rangle e_1') + (\langle\langle \uparrow\text{Int} \Leftarrow T_2\rangle\rangle e_2') : \downarrow\text{Int}$, by (IT-add)

$\square$

e	$::= n \mid \lambda x : T.e \mid \langle T \Leftarrow T \rangle\ e \mid e\ e \mid e + e$	Expressions
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	$::= T \rightarrow T \mid \downarrow F \mid \uparrow F$	Function type
T	$::= P \mid T \rightarrow T \mid \uparrow T \mid \downarrow T$	Type
E	$::= \square\ e \mid v\ \square \mid \square + e \mid v + \square \mid \langle T \Leftarrow T \rangle\ \square$	Evaluation Frames

Figure A.1: SCGT taint-tracking: Syntax.

A.3 SCGT: Correctness of Qualifiers

Taint-tracking internal language definition:

- syntax: Figure A.1
- static semantics: Figure A.2
- dynamic semantics: Figure A.3

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 5. *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), $T_1 <: \downarrow T_2$ $\square$

Lemma 6. *(Inversion, taint tracking)*

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 $T = \downarrow(T_1 \rightarrow T_2)$*
6. *if $\emptyset \vdash \lambda^{\widehat{\circ}} x : T'.e : T$, then $T = \updownarrow(T_1 \rightarrow T_2)$*

$$\text{(TTT-num1)}\frac{}{\Gamma \vdash n^{\circ} : \downarrow\text{Int}} \qquad \text{(TTT-num2)}\frac{}{\Gamma \vdash n^{\bullet} : \text{Int}}$$

$$\text{(TTT-var)}\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(TTT-err)}\frac{}{\Gamma \vdash \text{CastError} : T}$$

$$\text{(TTT-abs1)}\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda^{\circ} x : T_1.e : \downarrow(T_1 \to T_2)}$$

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

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

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

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

Figure A.2: SCGT taint-tracking: Typing.

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

$$\text{(TTE-app2)}\ \frac{T = {\uparrow}T'}{(\lambda x : T.e)\ v \longrightarrow e[\widehat{v}/x]} \qquad \text{(TTE-add)}\ \frac{n_3 = n_1 + n_2}{n_1 + n_2 \longrightarrow n_3^\circ}$$

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

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

$$\text{(TTE-remove1)}\ \frac{T_1 <: T_2 \qquad T_2 \neq {\uparrow}T'}{\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow v} \qquad \text{(TTE-remove2)}\ \frac{T_1 <: T_2 \qquad T_2 = {\uparrow}T'}{\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow \widehat{v}}$$

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

$$\text{(TTE-err)}\ \frac{}{E[\text{CastError}] \longrightarrow \text{CastError}}$$

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

Figure A.3: SCGT taint-tracking: Evaluation.

7. *if* $\emptyset \vdash \lambda^{\bullet} x : T'.e : T$*, then* $T = T_1 \rightarrow T_2$
8. *if* $\emptyset \vdash \lambda^{\widehat{\bullet}} x : T'.e : T$*, then* $T = \uparrow(T_1 \rightarrow T_2)$
9. *if* $\emptyset \vdash \langle F \Leftarrow F' \rangle\ f : T$*, then* $T = T_1 \rightarrow T_2$ *or* $T = \uparrow(T_1 \rightarrow T_2)$
10. *if* $\emptyset \vdash \langle Dyn \Leftarrow T \rangle b : T$*, then* $T = Dyn$

Proof. By case analysis:

Case v=n°

1. $\emptyset \vdash n^{\circ} : \downarrow$Int, by (TTT-num1)

Case v=$n^{\widehat{\circ}}$

1. $\emptyset \vdash n^{\circ} : \downarrow$Int, by (TTT-num1)
2. $\emptyset \vdash n^{\widehat{\circ}} : \updownarrow$Int, by (TTT-$\uparrow$)

Case v=$n^{\bullet}$

1. $\emptyset \vdash n^{\bullet}$: Int, by (TTT-num2)

Case v=$n^{\widehat{\bullet}}$

1. $\emptyset \vdash n^{\bullet}$: Int, by (TTT-num2)
2. $\emptyset \vdash n^{\widehat{\bullet}} : \uparrow$Int, by (TTT-$\uparrow$)

Case v=$\lambda^{\circ} x : T.e$

1. $\emptyset \vdash \lambda^{\circ} x : T.e : \downarrow(T \rightarrow T')$, by (TTT-abs1)

Case v=$\lambda^{\widehat{\circ}} x : T.e$

1. $\emptyset \vdash \lambda^{\circ} x : T.e : \downarrow(T \rightarrow T')$, by (TTT-abs1)
2. $\emptyset \vdash \lambda^{\widehat{\circ}} x : T.e : \updownarrow(T \rightarrow T')$, by (TTT-$\uparrow$)

Case v=$\lambda^{\bullet} x : T.e$

1. $\emptyset \vdash \lambda^{\bullet} x : T.e : T \rightarrow T'$, by (TTT-abs2)

Case v=$\lambda^{\widehat{\bullet}} x : T.e$

1. $\emptyset \vdash \lambda^{\bullet} x : T.e : T \rightarrow T'$, by (TTT-abs2)
2. $\emptyset \vdash \lambda^{\widehat{\bullet}} x : T.e : \uparrow(T \rightarrow T')$, by (TTT-$\uparrow$)

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. $\emptyset \vdash \langle F \Leftarrow F' \rangle f : F$, $|F| = T \rightarrow T'$ by definition of F
4. $F \not<: \downarrow(T \rightarrow T')$, by contradiction between assumptions and Lemma 5
5. $\emptyset \vdash \langle F \Leftarrow F' \rangle f : F$, $F = T \rightarrow T'$ or $F = \uparrow(T \rightarrow T')$

Case v=$\langle$Dyn $\Leftarrow T \rangle b$

1. $\emptyset \vdash \langle \text{Dyn} \Leftarrow T \rangle b : \text{Dyn}$, by (TTT-cast)

□

Lemma 7. *(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* $T <: \downarrow(T_1 \rightarrow T_2)$, *then* $\exists T', x, e,\ v = \lambda^{\circ} x : T'.e$ *and* $T <: T'$
5. $\emptyset \vdash v : T$ *and* $T <: T_1 \rightarrow T_2$, *then* $v = \lambda^{\circ} x : T'.e$ *or* $v = \lambda^{\bullet} x : T'.e$ *or* $v = \langle T_1 \rightarrow T_2 \Leftarrow F' \rangle f$
6. $\emptyset \vdash v : T$ *and* $T <: \uparrow(T_1 \rightarrow T_2)$, *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. By case analysis:

Case $T <: \downarrow$Int

1. $v = n^{\circ}$, by Lemma 6-1

Case $T <:$ Int

1. $v = n^{\circ}$ or $v = n^{\bullet}$, by Lemma 6-1,3

Case $T <: \uparrow$Int

1. $v = n^{\circ}$ or $v = n^{\widehat{\circ}}$ or $v = n^{\bullet}$ or $v = n^{\widehat{\bullet}}$, by Lemma 6-1,2,3,4
2. $v = n$

Case $T <: \downarrow(T_1 \rightarrow T_2)$

1. $v = \lambda^{\circ} x : T'.e$, by Lemma 6-5

Case $T <: T_1 \rightarrow T_2$

1. $v = \lambda^{\circ} x.e$ or $v = \lambda^{\bullet} x.e$ or $v = \langle F \Leftarrow F' \rangle f$, by Lemma 6-5,7,9
2. $v = \lambda^{\circ} x.e$ or $v = \lambda^{\bullet} x.e$ or $v = \langle T_1 \rightarrow T_2 \Leftarrow F' \rangle f$, by (TTT-cast)

Case $T <: \uparrow(T_1 \to T_2)$

1. $v = \lambda^{\circ}x.e$ or $v = \widehat{\lambda^{\circ}}x.e$ or $v = \lambda^{\bullet}x.e$ or $v = \widehat{\lambda^{\bullet}}x.e$ or $v = \langle F \Leftarrow F' \rangle f$, by Lemma 6-5,6,7,8,9
2. $v = f$

Case $T = \mathbf{Dyn}$

1. $v = \langle \text{Dyn} \Leftarrow T \rangle b$, by Lemma 6-10
2. $T \rightsquigarrow \text{Dyn}$, by (TTT-cast)
3. $T \neq \uparrow T'$
4. $b \neq \widehat{v}$, by Lemma 7-1,2,4,5

$\square$

Lemma 8. *If $\emptyset \vdash v : T$ and $T <: \uparrow T$, then $\emptyset \vdash \widehat{v} : \uparrow T$*

Proof. By case analysis of values:

Case $T = \mathbf{Int}$

1. $v = n$, by Lemma 7-3
2. $\emptyset \vdash \widehat{n} : \uparrow\text{Int}$, by (TTT-$\uparrow$)

Case $T = T_1 \to T_2$

1. $v = f$, by Lemma 7-6
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) $\langle \uparrow \widehat{T \Leftarrow T'} \rangle\ f : \uparrow T$, by (TTT-cast)

$\square$

Theorem 11. *(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-$\uparrow$)

1. e is a value

Case (TTT-err)

1. e is CastError

Case (TTT-var)

1. Case impossible with an empty environment (Well-typed)

Case (TTT-app)

1. By assumption:
   (a) $\emptyset \vdash e_1\ e_2 : T_3$
   (b) $\emptyset \vdash e_1 : T_1$, $\emptyset \vdash e_2 : T_2$, $|T_1| = T_{11} \rightarrow T_{12}$, $T_2 <: T_{11}$
2. If e_1 is not a value, use rule (TTE-congr) with $E = \Box\ e_2$ to progress.
3. If e_1 is CastError, use rule (TTE-err) with $E = \Box\ e_2$ to progress.
4. If e_1 is a value and e_2 is not a value, use rule (TTE-congr) with $E = e_1\ \Box$ to progress.
5. If e_1 is a value and e_2 is CastError, use rule (TTE-err) with $E = e_1\ \Box$ 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 7-6. 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 $E = \Box + e_2$ to progress.
3. If e_1 is CastError, use rule (TTE-err) with $E = \Box + e_2$ to progress.
4. If e_1 is a value and e_2 is not a value, use rule (TTE-congr) with $E = e_1 + \Box$ to progress.
5. If e_1 is a value and e_2 is CastError, use rule (TTE-err) with $E = e_1 + \Box$ to progress.
6. If e_1 and e_2 are both values, then $e_1 = n_1$ and $e_2 = n_2$ by Lemma 7-3. 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 $E = \langle T_2 \Leftarrow T_1 \rangle\ \Box$ to progress.
3. If e is CastError, use rule (TTE-err) with $E = \langle T_2 \Leftarrow T_1 \rangle\ \Box$ 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 7-7. 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 9. *(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) $\Gamma, x : T \vdash \lambda^{\circ} y : T'.e : \downarrow(T' \rightarrow T'')$
   (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. $\Gamma \vdash \lambda^{\circ} y : T'.(e[v/x]) : \downarrow(T' \rightarrow T'')$, by (TTT-abs1)
5. $\Gamma \vdash (\lambda^{\circ} y : T'.e)[v/x] : \downarrow(T' \rightarrow T'')$, by Substitution definition

Case (TTT-abs2)

1. By assumption:
   (a) $\Gamma, x : T \vdash \lambda^{\bullet} y : T'.e : T' \rightarrow T''$
   (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. $\Gamma \vdash \lambda^{\bullet} y : T'.(e[v/x]) : T' \rightarrow T''$, by (TTT-abs2)
5. $\Gamma \vdash (\lambda^{\bullet} y : T'.e)[v/x] : T' \rightarrow T''$, by Substitution definition

Case (TTT-$\uparrow$)

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 12. *(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) $\emptyset \vdash e_1\ e_2 : T_{12}$, $\emptyset \vdash e_1 : T_1$, $\emptyset \vdash e_2 : T_2$, $|T_1| = T_{11} \rightarrow T_{12}$, $T_2 <: T_{11}$
   (c) $e_1 \longrightarrow e_1'$
2. $\emptyset \vdash e_1' : T_1'$, $T_1' <: T_1$, $|T_1'| = T_{11}' \rightarrow T_{12}'$, 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) $\emptyset \vdash e_1\ e_2 : T_{12}$, $\emptyset \vdash e_1 : T_1$, $\emptyset \vdash e_2 : T_2$, $|T_1| = T_{11} \rightarrow T_{12}$, $T_2 <: T_{11}$
   (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. $T_1 <: \uparrow$Int,$T_2 <: \uparrow$Int,$T_3 <: \downarrow$Int (From TTT-add)
3. $T_1' <: T_1$, by induction on (1-c)
4. $T_1' <: \uparrow$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 8
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 $\text{Dyn} \not\rightsquigarrow \downarrow T_2'$
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 8
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 13. *($\downarrow$ correctness) If $\emptyset \vdash v : \downarrow T$, then $v = v'^{\circ}$.*

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

Theorem 14. *($\uparrow$ 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. □

Erasure. To define taint erasure, we use brown coloring to denote an expression that lives in the original (non-taint-tracking) language.

Definition 1. *(Erase)*

1. $erase(n) = n$
2. $erase(\lambda x : T.e) = \lambda x : T.e$
3. $erase(\langle T_2 \Leftarrow T_1\rangle\ e) = \langle T_2 \Leftarrow T_1\rangle\ erase(e)$
4. $erase(e_1\ e_2) = erase(e_1)\ erase(e_2)$
5. $erase(e_1 + e_2) = erase(e_1) + erase(e_2)$
6. $erase(e[v/x]) = erase(e)[erase(v)/x]$

Lemma 10. $erase(e) = e$

Proof. By induction on the syntax:

Case $e = n$

1. $\text{erase}(n) = n$, by Definition 1-1

Case $e = \lambda x : T.e$

1. $\text{erase}(\lambda x : T.e) = \lambda x : T.e$, by Definition 1-2

Case $e = \langle T_2 \Leftarrow T_1 \rangle\ e'$

1. $\text{erase}(\langle T_2 \Leftarrow T_1 \rangle\ e') = \langle T_2 \Leftarrow T_1 \rangle\ \text{erase}(e')$, by Definition 1-3
2. $\text{erase}(e') = e'$,by induction on e'
3. $\langle T_2 \Leftarrow T_1 \rangle\ \text{erase}(e') = \langle T_2 \Leftarrow T_1 \rangle\ e'$, by 2

Case $e = e_1\ e_2$

1. $\text{erase}(e_1\ e_2) = \text{erase}(e_1)\ \text{erase}(e_2)$, by Definition 1-4
2. $\text{erase}(e_1) = e_1$, $\text{erase}(e_2) = e_2$, by induction on e_1 and e_2
3. $\text{erase}(e_1)\ \text{erase}(e_2) = e_1\ e_2$, by 2

Case $e = e_1 + e_2$

1. $\text{erase}(e_1 + e_2) = \text{erase}(e_1){+}\text{erase}(e_2)$, by Definition 1-5
2. $\text{erase}(e_1) = e_1$, $\text{erase}(e_2) = e_2$, by induction on e_1 and e_2
3. $\text{erase}(e_1){+}\text{erase}(e_2) = e_1{+}e_2$, by 2

□

Corollary 1. *$erase(v) = v$*

Proof. Direct proof using Lemma 10 □

Theorem 15. *(Tainting faithfulness) If $e \longrightarrow e'$, then $erase(e) \longrightarrow erase(e')$.*

Proof. By induction on the evaluation rules.

Case (TTE-congr)

Subcase $E = \square\ e$

1. By assumption:
   (a) $e = e_1\ e_2$, $e' = e_1'\ e_2$
   (b) $e_1 \longrightarrow e_1'$
2. $\text{erase}(e_1\ e_2) = \text{erase}(e_1)\ \text{erase}(e_2)$, by Definition 1-4
3. $\text{erase}(e_1) \longrightarrow \text{erase}(e_1')$, by induction on (1-b)
4. $\text{erase}(e_1)\ \text{erase}(e_2) \longrightarrow \text{erase}(e_1')\ \text{erase}(e_2)$, by (E-congr) with $E = \square\ e$
5. $\text{erase}(e_1')\ \text{erase}(e_2) = \text{erase}(e_1'\ e_2)$, by Definition 1-4

Subcase $E = v\,\Box$ Analogous to subcase $E = \Box\, e$

Subcase $E = \Box + e$

1. By assumption:
   (a) $e = e_1 + e_2$, $e' = e'_1 + e_2$
   (b) $e_1 \longrightarrow e'_1$
2. $\text{erase}(e_1 + e_2) = \text{erase}(e_1) + \text{erase}(e_2)$, by Definition 1-5
3. $\text{erase}(e_1) \longrightarrow \text{erase}(e'_1)$, by induction on (1-b)
4. $\text{erase}(e_1) + \text{erase}(e_2) \longrightarrow \text{erase}(e'_1) + \text{erase}(e_2)$, by (E-congr) with $E = \Box + e$
5. $\text{erase}(e'_1) + \text{erase}(e_2) = \text{erase}(e'_1 + e_2)$, ¿by Definition 1-5

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

Subcase $E = \langle T \Leftarrow T\rangle\, \Box$

1. By assumption:
   (a) $e = \langle T_2 \Leftarrow T_1\rangle\, e''$, $e' = \langle T_2 \Leftarrow T_1\rangle\, e'''$
   (b) $\emptyset \vdash T_2 \Leftarrow T_1\rangle\, e' : T_2$, $\emptyset \vdash e' : T_1$
   (c) $e'' \longrightarrow e'''$
2. $\text{erase}(\langle T_2 \Leftarrow T_1\rangle\, e'') = \langle T_2 \Leftarrow T_1\rangle\, \text{erase}(e'')$, by Definition 1-3
3. $\text{erase}(e'') \longrightarrow \text{erase}(e''')$ by induction of (1-c)
4. $\langle T_2 \Leftarrow T_1\rangle\, \text{erase}(e'') \longrightarrow \langle T_2 \Leftarrow T_1\rangle\, \text{erase}(e''')$ by (E-congr) with $E = \langle T \Leftarrow T\rangle\, \Box$
5. $\langle T_2 \Leftarrow T_1\rangle\, \text{erase}(e''') = \text{erase}(\langle T_2 \Leftarrow T_1\rangle\, e''')$, by Definition 1-3

Case (TTE-app1)

1. By assumption:
   (a) $e = (\lambda x : T.e'')\, v$, $e' = e''[v/x]$
   (b) $T \neq \uparrow T'$
2. $\text{erase}((\lambda x : T.e'')\, v) = \text{erase}(\lambda x : T.e'')\, \text{erase}(v)$
3. $\text{erase}(\lambda x : T.e'')\, \text{erase}(v) = (\lambda x : T.e'')\, v$, by Definition 1-2 and Lemma 10
4. $(\lambda x : T.e'')\, v \longrightarrow e''[v/x]$, by (E-app)
5. $\text{erase}(e''[v/x]) = \text{erase}(e'')[\text{erase}(v)/x]$, by Definition 1-6
6. $\text{erase}(e'')[\text{erase}(v)/x] = e''[v/x]$, by Lemma 10 on both e'' and v

Case (TTE-app2)

1. By assumption:

   (a) $e = (\lambda x : T.e'')\ v,\ e' = e''[\widehat{v}/x]$

   (b) $(\lambda x : T.e)\ v \longrightarrow e[\widehat{v}/x]$

   (c) $T = \uparrow T'$

2. $\text{erase}((\lambda x : T.e'')\ v) = \text{erase}(\lambda x : T.e'')\ \text{erase}(v)$

3. $\text{erase}(\lambda x : T.e'')\ \text{erase}(v) = (\lambda x : T.e'')\ v$, by Definition 1-2 and Lemma 10

4. $(\lambda x : T.e'')\ v \longrightarrow e''[v/x]$, by (E-app)

5. $\text{erase}(e''[\widehat{v}/x]) = \text{erase}(e'')[\text{erase}(\widehat{v})/x]$, by Definition 1-6

6. $\text{erase}(e'')[\text{erase}(\widehat{v})/x] = e''[v/x]$, by Lemma 10 on both e'' and $\widehat{v}$

Case (TTE-add)

1. By assumption:

   (a) $e = n_1 + n_2,\ e' = n_3^\circ$

   (b) $n_3 = n_1 + n_2$

2. $\text{erase}(n_1 + n_2) = \text{erase}(n_1) + \text{erase}(n_2)$, by Definition 1-5

3. $\text{erase}(n_1) + \text{erase}(n_2) = n_1 + n_2$, by Definition 1-1

4. $n_1 + n_2 \longrightarrow n_3$, by (E-add)

5. $\text{erase}(n_3^\circ) = n_3$, by Definition 1-1

Case (TTE-merge1)

1. By assumption:

   (a) $e = \langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle\ v^\circ, e' = \langle T_2 \Leftarrow |T_1|\rangle\ v^\bullet$

   (b) $T_1 \rightsquigarrow T_2$

2. $\text{erase}(\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle\ v^\circ) = \langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle\ \text{erase}(v^\circ)$, by Definition 1-3

3. $\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle\ \text{erase}(v^\circ) = \langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle\ v$, by Lemma 10

4. $\langle T_2 \Leftarrow \text{Dyn}\rangle\langle \text{Dyn} \Leftarrow T_1\rangle\ v \longrightarrow \langle T_2 \Leftarrow |T_1|\rangle\ v$, by (E-merge)

5. $\text{erase}(\langle T_2 \Leftarrow |T_1|\rangle\ v^\bullet) = \langle T_2 \Leftarrow |T_1|\rangle\ \text{erase}(v^\bullet)$, by Definition 1-3

6. $\langle T_2 \Leftarrow |T_1|\rangle\ \text{erase}(v^\bullet)$
   $= \langle T_2 \Leftarrow |T_1|\rangle\ v$, by Lemma 10

Case (TTE-merge2)

1. By assumption:
   (a) $e = \langle T_2 \Leftarrow \mathrm{Dyn}\rangle\langle \mathrm{Dyn} \Leftarrow T_1\rangle\ v^\bullet$,$e' = \langle T_2 \Leftarrow T_1\rangle\ v^\bullet$
   (b) $T_1 \rightsquigarrow T_2$
2. $\mathrm{erase}(\langle T_2 \Leftarrow \mathrm{Dyn}\rangle\langle \mathrm{Dyn} \Leftarrow T_1\rangle\ v^\bullet) =$
   $\langle T_2 \Leftarrow \mathrm{Dyn}\rangle\langle \mathrm{Dyn} \Leftarrow T_1\rangle\ \mathrm{erase}(v^\bullet)$, by Definition 1-3
3. $\langle T_2 \Leftarrow \mathrm{Dyn}\rangle\langle \mathrm{Dyn} \Leftarrow T_1\rangle\ \mathrm{erase}(v^\bullet) =$
   $\langle T_2 \Leftarrow \mathrm{Dyn}\rangle\langle \mathrm{Dyn} \Leftarrow T_1\rangle\ v$, by Lemma 10
4. $\langle T_2 \Leftarrow \mathrm{Dyn}\rangle\langle \mathrm{Dyn} \Leftarrow T_1\rangle\ v \longrightarrow \langle T_2 \Leftarrow |T_1|\rangle\ v$, by (E-merge)
5. $T_1 \neq \uparrow T_1'$, because $T_1 \not\rightsquigarrow \mathrm{Dyn}$
6. $T_1 = |T_1|$, by Lemma 2-7
7. $\mathrm{erase}(\langle T_2 \Leftarrow T_1\rangle\ v^\bullet) = \mathrm{erase}(\langle T_2 \Leftarrow |T_1|\rangle\ v^\bullet)$, by replacing T_1
8. $\mathrm{erase}(\langle T_2 \Leftarrow |T_1|\rangle\ v^\bullet) = \langle T_2 \Leftarrow |T_1|\rangle\ \mathrm{erase}(v^\bullet)$, by Definition 1-3
9. $\langle T_2 \Leftarrow |T_1|\rangle\ \mathrm{erase}(v^\bullet) = \langle T_2 \Leftarrow |T_1|\rangle\ v$, by Lemma 10

Case (TTE-remove1)

1. By assumption:
   (a) $e = \langle T_2 \Leftarrow T_1\rangle\ v$, $e' = v$
   (b) $T_1 <: T_2$, $T_2 \neq \uparrow T_2'$
2. $\mathrm{erase}(\langle T_2 \Leftarrow T_1\rangle\ v) = \langle T_2 \Leftarrow T_1\rangle\ \mathrm{erase}(v)$, by Definition 1-3
3. $\langle T_2 \Leftarrow T_1\rangle\ \mathrm{erase}(v) = \langle T_2 \Leftarrow T_1\rangle\ v$, by Lemma 10
4. $\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow v$, by (E-remove)
5. $\mathrm{erase}(v) = v$, by Lemma 10

Case (TTE-remove2)

1. By assumption:
   (a) $e = \langle T_2 \Leftarrow T_1\rangle\ v$, $e' = \widehat{v}$
   (b) $T_1 <: T_2$, $T_2 = \uparrow T_2'$
2. $\mathrm{erase}(\langle T_2 \Leftarrow T_1\rangle\ v) = \langle T_2 \Leftarrow T_1\rangle\ \mathrm{erase}(v)$, by Definition 1-3
3. $\langle T_2 \Leftarrow T_1\rangle\ \mathrm{erase}(v) = \langle T_2 \Leftarrow T_1\rangle\ v$, by Lemma 10
4. $\langle T_2 \Leftarrow T_1\rangle\ v \longrightarrow v$, by (E-remove)
5. $\mathrm{erase}(\widehat{v}) = v$, by Lemma 10

Case (TTE-fcastinv)

1. By assumption:
   (a) $e = (\langle F' \Leftarrow F\rangle\ f)\ v$, $e' = \langle T_2' \Leftarrow T_2\rangle(f\ (\langle T_1 \Leftarrow T_1'\rangle\ v))$
   (b) $|F| = T_1 \to T_2$, $|F'| = T_1' \to T_2'$, $F \rightsquigarrow F'$, $F \not<: F'$
2. erase$((\langle F' \Leftarrow F\rangle\ f)\ v) = (\langle F' \Leftarrow F\rangle\ f)\ v$, by Lemma 10
3. $(\langle F' \Leftarrow F\rangle\ f)\ v \longrightarrow \langle T_2' \Leftarrow T_2\rangle(f\ (\langle T_1 \Leftarrow T_1'\rangle\ v))$, by (E-fcastinv)
4. erase$(\langle T_2' \Leftarrow T_2\rangle(f\ (\langle T_1 \Leftarrow T_1'\rangle\ v))) =$
   $\langle T_2' \Leftarrow T_2\rangle(f\ (\langle T_1 \Leftarrow T_1'\rangle\ v))$, by Lemma 10

□

A.4 RCGT- Type Safety

Progress is the same as SCGT.

Theorem 16. *(Preservation) If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T'$ and $T' <: T$.*

Proof. Same as SCGT, except:

Case (E-merge)

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

□

A.5 RCGT- Correctness of Qualifiers

Lemma 11. *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 $T_1 \neq \text{Dyn}$ 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 12. *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 17. *(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 12, $T_1 <: T_2$.
2. However, by definition of value, $T_1 \not<: T_2$. $\Rightarrow\Leftarrow$

Case $T_2 = \downarrow T_2'$

1. By Lemma 11, $T_1 <: T_2$.
2. However, by definition of value, $T_1 \not<: T_2$. $\Rightarrow\Leftarrow$

□

Appendix B

Confined Gradual Typing: Microbenchmark Results

In these tables, time taken is reported in milliseconds and size means multiples of 100,000 collection elements. The maximum measured relative error in the microbenchmarks is ±13.48%, with a confidence level of 95%.

Size	GT	RCGT	RCGT with $\downarrow$	RCGT with $\uparrow$
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 B.1: 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 B.2: 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 B.3: 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 B.4: 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 B.5: Closure evaluation, Wrapping benchmark.

List of Figures

Bibliography

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991. 4

[2] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 96(1):52–69, Dec. 2014. 8, 22, 60, 106

[3] E. Allende, J. Fabry, R. Garcia, and É. Tanter. Confined gradual typing. In *Proceedings of the 29th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2014)*, pages 251–270, Portland, OR, USA, Oct. 2014. ACM Press. 8, 10, 75, 107

[4] 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). 8, 53, 106

[5] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 53–64, 2007. 51

[6] C. Anderson, S. Drossopoulou, and P. Giannini. Towards type inference for javascript. In A. P. Black, editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, number 3586 in Lecture Notes in Computer Science, pages 428–452, Glasgow, UK, July 2005. Springer-Verlag. 51

[7] 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. 20

[8] G. Bracha. The strongtalk type system for smalltalk. http://www.bracha.org/nwst.html. 49

[9] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, pages 1–6, 2004. 20, 49

[10] 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). 29, 49

[11] O. Callaú. *Empirically-Driven Design and Implementation of Gradualtalk.* PhD thesis, University of Chile, 2014. 22

[12] O. Callaú and É. Tanter. Programming with ghosts. *IEEE Software*, 30(1):74–80, 2013. 46

[13] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997. 3, 41

[14] 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. 19

[15] 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. 149, 150, 151

[16] M. Chang, B. Mathiske, E. Smith, A. Chaudhuri, A. Gal, M. Bebenita, C. Wimmer, and M. Franz. The impact of optional type information on JIT compilation of dynamically-typed languages. In *Proceedings of the ACM Dynamic Languages Symposium (DLS 2007)*, pages 13–24, Montreal, Canada, Oct. 2007. ACM Press. 18

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999. 1

[18] Dart Team. Dart programming language specification, May 2013. Version 0.41. 73

[19] É. P. F. de Lausanne (EPFL). Scala. http://www.scala-lang.org. 37

[20] D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 3.12*. Institut National de Recherche en Informatique et en Automatique, http://caml.inria.fr/pub/docs/manual-ocaml/index.html, Jul 2011. 37

[21] I. Figueroa, É. Tanter, and N. Tabareau. A practical monadic aspect weaver. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*, pages 21–26, Potsdam, Germany, Mar. 2012. ACM Press. 50

[22] 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. 18, 31, 78

[23] T. Freeman and F. Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. 20

[24] M. Furr. *Combining Static and Dynamic Typing in Ruby*. PhD thesis, University of Maryland, 2009. 50

[25] R. Garcia. Calculating threesomes, with blame. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 417–428, New York, NY, USA, 2013. ACM. 19

[26] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983. 23

[27] J. O. Graver and R. E. Johnson. A type system for smalltalk. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 136–150, 1990. 50, 51

[28] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, Nov. 2000. 91

[29] N. Haldiman, M. Denker, and O. Nierstrasz. Practical, pluggable types for a dynamic language. *Comput. Lang. Syst. Struct.*, 35(1):48–62, 2009. 50, 51

[30] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher-Order and Sympolic Computation*, 23(2):167–189, June 2010. 17, 53, 104

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. 1

[32] 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. 15, 32, 33, 41

[33] R. E. Johnson. Type-checking smalltalk. *SIGPLAN Not.*, 21(11):315–321, 1986. 50, 51

[34] R. E. Johnson, J. O. Graver, and L. W. Zurawski. Ts: an optimizing compiler for smalltalk. *SIGPLAN Not.*, 23(11):18–26, 1988. 50, 51

[35] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2):Article n.6, Jan. 2010. 20

[36] S. Krishnamurthi. *Programming Languages: Application and Interpretation*. 2007. Version 2007-04-26. 3

[37] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In J. Vitek, editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 260–284, Paphos, Cyprus, July 2008. Springer-Verlag. 37

[38] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In Castagna [15], pages 95–111. 37

[39] 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. 78, 104

[40] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. 1

[41] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. 1, 2, 32, 34, 35, 36, 41

[42] F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, 2009. 50, 51

[43] *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2010)*, Madrid, Spain, Jan. 2010. ACM Press. 150, 151

[44] 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, 51, 53, 104

[45] D. Rémy. Theoretical aspects of object-oriented programming. chapter Type Inference for Records in Natural Extension of ML, pages 67–95. MIT Press, Cambridge, MA, USA, 1994. 16

[46] B. V. Ryseghem, S. Ducasse, and J. Fabry. Seamless composition and reuse of customizable user interfaces with Spec. *Science of Computer Programming*, 96(1):34–51, Dec. 2014. 101

[47] C. Saito and A. Igarashi. Self type constructors. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 263–282, New York, NY, USA, 2009. ACM. 29, 33, 41

[48] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In L. Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 248–274, Darmstadt, Germany, July 2003. Springer-Verlag. 36

[49] J. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In Castagna [15], pages 17–31. 17, 18, 19, 31, 104

[50] J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006. 4, 5, 10, 11, 12, 13, 15, 57, 84, 85, 86

[51] 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. 4, 5, 10, 12, 13, 15, 37, 39, 42, 54, 57, 58, 74, 95

[52] J. Siek and P. Wadler. Threesomes, with and without blame. In POPL 2010 [43], pages 365–376. 18, 19, 53, 104

[53] R. Smit. Pegon. http://sourceforge.net/projects/pegon/. 49

[54] 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. 17, 104, 109

[55] 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. 16, 72

[56] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, Aug. 1976. 1

[57] S. Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs.* PhD thesis, Northeastern University, Jan. 2010. 18, 23, 49, 103

[58] 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. 18, 78, 104

[59] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press. 53

[60] S. Tobin-Hochstadt and V. St-Amour. The typed racket guide. http://docs.racket-lang.org/ts-guide/. 49

[61] L. Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, Jul 2009. 2, 3, 5

[62] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In Castagna [15], pages 1–16. 18, 31, 39, 78, 104

[63] T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In POPL 2010 [43], pages 377–388. 4, 20, 53