RICE UNIVERSITY

Designing Type Inference for Typed Object-Oriented Languages

by

Daniel Smith

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Robert Cartwright, Chair
Professor of Computer Science

Walid Taha
Assistant Professor of Computer Science

Richard Price
Assistant Professor of Accounting

Eric Allen
Principal Investigator, Sun Labs

Houston, Texas

May, 2010

Abstract

Designing Type Inference for Typed Object-Oriented Languages

by

Daniel Smith

Type-checked object-oriented languages have typically been designed with extremely simple type systems. However, there has recently been intense interest in extending such languages with more sophisticated types and subtyping relationships. JAVA and C# are mainstream languages that have been successfully extended with generic classes and methods; SCALA, FORTRESS, and X10 are new languages that adopt more advanced typing features, such as arrows, tuples, unions, intersections, dependent types, and existentials.

Presently, the type inference performed by these languages is unstable and evolving. This thesis explores problems arising in the design of a type inference specification for such languages.

We first present a formal description of subtyping in the context of a variety of advanced typing features. We then demonstrate how our formal subtyping algorithm can be easily re-expressed to produce a type inference algorithm, and observe that this algorithm is general enough to address a variety of important type-checking problems.

Finally, we apply this theory to a case study of the JAVA language's type system. We express JAVA's types and inference algorithm in terms of our formal theory and note a variety of opportunities for improvement. We then describe the results of applying an improved type inference implementation to a selection of existing JAVA code, noting that, without introducing significant backwards-incompatibility problems for these programs, we've managed to significantly reduce the need for annotated method invocations.

Acknowledgments

A lot of significant life accomplishments seem to come easily (with patience), but completing a PhD thesis is not one of them!

My wife, Tara, has patiently endured years of graduate student life and, to finish the task, months of carrying much more than her share. This is something we've accomplished together.

Corky Cartwright, as my advisor, has been wonderfully supportive in encouraging me to explore the problems I find interesting and to let them take me wherever they might lead. Walid Taha and Eric Allen have had a significant role in directing my interests, by teaching good principles, sharing good ideas, and asking good questions. Richard Price has been kindly supportive through the years.

It's always rewarding to be surrounded by smart people, and Rice University has provided such an environment; I appreciate all I've gained from teachers, classmates, and colleagues.

LogicBlox, my current employer, has also been graciously supportive as I've pushed through the last few months of research and writing.

Finally, I owe a great debt to those who have, over many years, encouraged me to embrace learning and ignore perceived limits to what I can achieve—my parents, many teachers at all levels, and some close college friends. Most of them knew very little about Computer Science, but a lot about the value of knowledge and aspirations.

Contents

Chapter 1

Introduction

1.1 Context

Computer programs are instructions that describe how a computer should map input—something typed on a keyboard, gestures performed by a mouse, data stored in a file, etc.—to output—text or images displayed on a monitor, data stored in a file, etc. The standard conventions followed to express a program are called a *programming language*; the *semantics* of a particular language are a generic description of the steps that must be performed to map a program and its input to its output. If the semantics don't make sense for a particular program–input pair, an error occurs.

Certain programming languages are designed so that some potential errors can be recognized in a program independent of its input. This makes possible *static analysis*, the checking of a program for errors when it is written rather than when it is executed. This is useful for language designers, because it means that the language semantics can be based upon limiting assumptions about the programs they execute. It is useful for programmers, because it allows them to get immediate feedback when they've made a mistake.

One particularly fruitful kind of static analysis is *type checking*. Most languages contain *variables* and *expressions* which can be used to abstractly describe *values*.

For different inputs, an expression may represent a different value, but usually those different values all have similar properties, and can appear interchangably in other expressions. Thus, it's useful to assign a *type* to each expression, which describes the set of values that the expression may represent. Type checking verifies that expressions with particular types only appear in contexts in which values of that type make sense. A *type system* is a portion of the language definition that describes types and how they are used in static analysis.

For example, some "object-oriented" languages allow programs to define *objects,* a kind of value that bundles together data about some entity and various functions for operating on it. In this context, a type might be a description of the functions bundled by an object. The type checker would be used to verify that the program never tries to apply a function that an object might not contain.

In a typical object-oriented language, expressions can have many types—a program can declare certain classes of objects as "extensions" of other classes of objects, for example, thus forming hierarchies or directed graphs in which an object has a number of "ancestor" types, each providing a less-specific description of the object. To manage this complexity, type checkers usually determine just one *minimal* (or most-specific) type for each expression; then when the type checker needs to guarantee that some expression has type T, and given that the expression has minimal type S, the type checker verifies that S is a *subtype* of T.

To document programmers' intent and help guide type checking, type-checked

languages typically make use of *type annotations*, in which part of the program is a description of the types expected to appear in a particular context. In some cases, these annotations are important for type checking, but tedious for programmers to write and maintain. So a language might support *type inference*, a process by which the type checker infers the appropriate type annotations in cases in which they were elided. In languages with subtyping, type inference and subtyping are closely linked.

This thesis focuses on the design of type systems for one particular class of programming languages: type-checked, object-oriented languages that make use of subtyping and type inference.

1.2 Purpose

Type-checked object-oriented languages have typically been designed with extremely simple type systems: class declarations introduce types, and relationships between types are explicitly stated. However, there has recently been intense interest in extending this paradigm with more sophisticated types and subtyping relationships. JAVA and C# are mainstream languages that have been successfully extended with generic classes and methods; SCALA, FORTRESS, and X10 are new languages that adopt more advanced typing features, such as arrows, tuples, unions, intersections, dependent types, and existentials. All of these additional features allow more programmer expressiveness, but the burden of complexity quickly dictates that some form of inference take place, allowing programmers to elide some type annotations.

Presently, the type inference performed in these languages is unstable and evolving—in JAVA, inference is inconsistent among implementations and poorly-specified; in FORTRESS and X10, a concrete inference specification has not yet been produced. This thesis explores problems arising in the design of such a specification.

1.3 Overview

In chapter 2, we explore the theory of subtyping, starting with a basic language of types and extending it with features that are relevant to object-oriented languages with advanced type systems. Chapter 3 establishes a formal framework for type inference driven by the subtype relation. Chapter 4 applies the theory established in the previous chapters to a case study of the JAVA language's type system, noting a variety of opportunities for improvement, and discussing how such changes would impact legacy code.

Chapter 2

Theory of Subtyping

The subtype relation is fundamental in most object-oriented languages' type systems. In this chapter, we'll establish a formal theory for modeling subtyping with a variety of advanced typing features. In each case, we'll consider how the subtype relation can be extended to include the new feature—first by describing the relation using straightforward *declarative* inference rules, and then by reexpressing the relation *algorithmically.*[1] This translation from a declarative to an algorithmic definition is important for two reasons: first, because it allows us to examine some issues that arise in a concrete implementation of subtype testing; and second, because the formal presentation of type inference in chapter 3 builds upon the algorithmic version of subtyping.

To limit our scope, there are a variety of important research endeavors that are *not* undertaken in this formal presentation:

- This is not a *denotational semantics* for types. We rely frequently on the intuition that types "represent" sets of values, and that the subtype and subset relations are similar. However, this correspondence is not explored formally; instead, we take an *operational* approach: types are, formally, syntactic entities, and the subtype relation is simply the set of pairs that can be shown to be

[1] We follow Pierce's methodology here, and adopt his terminology [14].

related by some application of inference rules.

- This is not a formal proof of *type safety* or *soundness and completeness*. Our ability to make conclusive statements about a particular type system is limited by our abstract discussion: rather than focus on a particular language, we address typing features that may be useful in a variety of contexts. So there is no attempt to describe or prove properties about the semantics of a particular language. In addition, we do not attempt to formally demonstrate the correspondence between the declarative and algorithmic subtyping definitions.

- This is not a *comprehensive* list of typing features. The features addressed in this section are drawn from concrete examples in real production or prototype object-oriented languages; an effort is made to avoid language-specific quirks and undue limiting assumptions. But these features are only a sample, meant to provide a flavor of the kind of work that would be done in developing subtyping and type inference for a concrete language.

- This is not a guide to *implementation*. While we occasionally mention how certain simplifications might help an implementation's performance, our focus is on the *specification* of type systems. Producing an implementation is a separate, complex problem.

2.1 Core Type System

To begin, we'll specify a simple core language of types and define a subtype relation over those types.

2.1.1 Types

Intuitively, a type represents a set of values. Claiming that an expression has a type T implies that, *if* the expression can be evaluated successfully, the result will be a value in the set represented by T.

$$\begin{array}{rcl} T & ::= & B \\ & & \top \\ & & \bot \end{array}$$

For now, the types we'll consider are all atomic—that is, they are not composed of other arbitrary types—and fall into three classes:

- The set of *base types*, B. The meaning of these types is language-specific (they may, for example, either be primitives or be declared in a particular program).
- The *top type*, $\top$, which represents the set of all values. All expressions have this type.
- The *bottom type*, $\bot$, which represents the empty set of values. If an expression has this type, it must always fail to evaluate successfully.

Typically, base types are represented as a set of names (although we do not preclude more complex structures). Throughout this thesis, we will treat names as globally-unique identifiers. The details of mapping the text of a program to these globally unique names (or guaranteeing that such a mapping is unnecessary) is beyond our scope. By making this assumption, we avoid the tedious details of variable shadowing and other name-related issues.

2.1.2 Declarative Subtyping

The *subtype* relation provides an ordering for types, from more specific to more general. Just as types intuitively represent sets, the subtype relation (between types) intuitively corresponds to the *subset* relation (betweeen sets).[2]

To define subtyping for an open set of base types, we'll need to express it in terms of a *type environment* Γ.[3] The environment contains the following relation:

> Γ.extends(A, B) asserts that base type A is a subtype of base type B. The details about which base types appear in this relation are specific to a particular language; of course, we require that the values of the

[2]It should be emphasized, however, that this correspondence is only an intuition, not a formal part of the definition. Indeed, the definition of subtyping in a particular language may not be strong enough to include certain pairs of types that are provably in the subset relation.

[3]Throughout this thesis, we'll use type environments to represent a variety of facts about the context in which a type is to be interpreted. In general, the environment, Γ, is a record grouping together various relations containing relevant facts. Each relation is referenced with dot notation—Γ.extends, for example, is the relation we'll be using here.

subtype actually belong to the supertype as well. This relation need not be reflexive or transitive, and may be infinite; however, the number of types extended by a particular type must be finite.

We can now define subtyping among our core types with the following inference rules. We'll call this a *declarative* subtyping definition, in contrast to the *algorithmic* definition in the next section.

$$\Gamma \vdash T \preceq T \text{ (Reflex)}$$

$$\frac{\Gamma \vdash S \preceq U, \Gamma \vdash U \preceq T}{\Gamma \vdash S \preceq T} \text{ (Trans)}$$

$$\frac{\Gamma.\text{extends}(A, B)}{\Gamma \vdash A \preceq B} \text{ (Base)}$$

$$\Gamma \vdash S \preceq \top \text{ (Top)}$$

$$\Gamma \vdash \bot \preceq T \text{ (Bottom)}$$

The rules Reflex and Trans guarantee reflexivity and transitivity, essential properties of any subtype relation. Next, the Base rule maps entries in Γ.extends into the subtype relation. Finally, the rules Top and Bottom define $\top$ as a supertype[4] and $\bot$ as a subtype of all types.

[4]The *supertype* relation is the inverse of *subtype*. It can be written $\succeq$.

2.1.3 Algorithmic Subtyping

In order for a type checker to test that one type is a subtype of another, the subtyping definition from the previous section must be reexpressed.[5] In particular, there is not a straightforward way to check the TRANS rule:

$$\frac{\Gamma \vdash S \preceq U, \Gamma \vdash U \preceq T}{\Gamma \vdash S \preceq T} \text{ (TRANS)}$$

Given a certain S and T, the rule provides no guidance on how an algorithm might find a suitable choice for U or, just as importantly, conclude that no such U exists.

As an alternative, we'll rewrite the BASE rule in a way that supports checking subtyping between base types without any need for TRANS:

$$\frac{\Gamma.\text{extends}(A, B), \Gamma \vdash B \preceq T}{\Gamma \vdash A \preceq T} \text{ (BASE*)}$$

Like TRANS, BASE* tests subtyping by checking for the existence of some third type satisfying a condition; but, unlike TRANS, it's clear where this type comes from: it's listed in Γ.extends.

To guarantee termination (because Γ.extends may contain cycles), we'll also need a relation in Γ for tracking recursive invocations:

[5] While this line of discussion may seem tedious as it applies to the core language of types, the process of reexpressing subtyping algorithmically will be less obvious and more important as additional typing features are introduced.

Γ.without(φ) requires that assertions about types be made without relying on the fact stated by φ (for our purposes currently, this fact always takes the form of a subtyping assertion).

Let $\Gamma \vdash S \preceq: T$ represent an invocation of the subtyping algorithm for types S and T in environment Γ. It can be resolved as follows:

1. If Γ.without contains "$S \preceq: T$" then the result is false.

2. Otherwise, a finite set of tests, as determined by the structures of S and T, and as outlined in the table below, are performed. The result is true if and only if one of these tests has a true result.

   In the table, an inference rule name represents a test that i) the corresponding rule conclusion matches S and T; and ii) the corresponding rule premise, altered by substituting Γ' for Γ, holds. Γ' is derived by extending Γ with the assertion Γ.without($S \preceq: T$).

		T		
		$\top$	$\bot$	B
S	$\top$	Top	-	-
	$\bot$	Top	Bottom	Bottom
	B	Top	-	Reflex, Base*

   For now, the only interesting case in the table is when both S and T are base types—in that situation, the algorithm first checks that they are the same, and,

if not, next recurs on any types that S extends. As we add typing features, this table will become more complex.

2.2 Unions and Intersections

With the type system described in the previous section as a basis, we now explore the impact of extending the system with some additional important classes of types.

It's worth pointing out that some degree of freedom for new features already exists in the core type system. Simple parameterized classes, for example, can just be treated as templates for generating base types; type variables bound by base types can themselves be encoded as base types in Γ. In contrast, the features we consider in the rest of this chapter cannot be adequately expressed (in full generality) by the extends relation.

To start, we'll extend the core type system with *union* and *intersection* types. Often in type checking it is useful to assert that an expression has *one* or *all* of a set of types. Type systems sometimes define complex *join* or *meet* functions that produce types conveying these constraints; a simpler and more powerful way to make these assertions is with union and intersection types.

2.2.1 Types

We extend the definition of types in section 2.1.1 with the following:

$$T \quad ::= \quad \ldots$$

$$\bigcup \overline{T}$$

$$\bigcap \overline{T}$$

- A *union type*, $\bigcup \overline{T}$,[6] which, naturally, represents the union of the sets corresponding to the given types. Concrete examples typically use infix notation, like $S \cup T$, which can be read "S union T" or "S or T."

- An *intersection type*, $\bigcap \overline{T}$, which, also naturally, represents the intersection of the sets corresponding to the given types. Concrete examples typically use infix notation, like $S \cap T$, which can be read "S intersect T" or "S and T."

2.2.2 Declarative Subtyping

We extend the definition of subtyping in section 2.1.2 with the following:

$$\frac{\forall i, \Gamma \vdash S_i \preceq T}{\Gamma \vdash \bigcup \overline{S} \preceq T} \ (\cup\text{-Super})$$

$$\Gamma \vdash T_i \preceq \bigcup \overline{T} \ (\cup\text{-Sub})$$

$$\Gamma \vdash \bigcap \overline{T} \preceq T_i \ (\cap\text{-Super})$$

[6] The notation $\overline{T}$ represents a (possibly-empty) list of types. For example, $\bigcup(\top, \bot)$ is a union type. While unions and intersections could be similarly defined in terms of *sets* of types, the decision to use lists here is important for ensuring determinism in some inference algorithms.

$$\frac{\forall i, \Gamma \vdash S \preceq T_i}{\Gamma \vdash S \preceq \bigcap \overline{T}} \ (\cap\text{-Sub})$$

$$\Gamma \vdash S \cap (\bigcup \overline{T}) \preceq \bigcup (S \cap T_1 \ldots S \cap T_n) \ (\cap\text{-}\cup\text{-Dist})$$

Unions and intersections are complementary, and these rules reflect that correspondence. The elements of a union are subtypes of the union; the elements of an intersection are supertypes of the intersection. A type is a supertype of a union if it is a shared supertype of all the elements; similarly, a type is a subtype of an intersection if it is a shared subtype of all the elements.

Some subtyping relationships between unions and intersections can't be shown by simply decomposing the types. The rule $\cap$-$\cup$-Dist, for example, can be used to show that an intersection is a subtype of a union if the elements of the union appropriately distribute the type information expressed by the intersection.

2.2.3 Equivalence Rules

With the introduction of unions and intersections, we've made it possible to express the same set of values in a variety of ways. If two types correspond to the same set of values, it is useful to consider them equivalent and intuitively interchangable. We can formally define type equivalence, $\simeq$, as follows:

$$\frac{\Gamma \vdash S \preceq T, \Gamma \vdash T \preceq S}{\Gamma \vdash S \simeq T} \text{ (Type-Equiv)}$$

Using the subtyping rules defined in section 2.2.2, we can prove the following important equivalences. Recognizing these relationships is useful when reasoning about types; it also helps to informally demonstrate the fundamental soundness of our subtyping definition.

For brevity, the environment Γ is elided from the equivalences expressed below, and a proof demonstrating the equivalence is not given. However, we do note which inference rules would be important in such a proof (taking for granted that Trans will often be used as well).

Special Unions and Intersections. Some important equivalences hold for nullary (empty) and unary (singleton) unions, and similarly for intersections.

$$\bot \simeq \bigcup() \text{ (Bottom, } \cup\text{-Super)}$$

$$\top \simeq \bigcap() \text{ } (\cap\text{-Sub, Top})$$

$$T \simeq \bigcup(T) \simeq \bigcap(T) \text{ } (\cup\text{-Sub, } \cup\text{-Super, etc.})$$

Interestingly, these equivalences mean that we could eliminate $\top$ and $\bot$ from our language of types, using empty unions and intersections instead. For clarity, however, we prefer to keep $\top$ and $\bot$.

Commutativity, Associativity, and Distributivity. As one might expect, the type operators $\cup$ and $\cap$ are commutative and associative; in addition, $\cup$ distributes over $\cap$ and vice versa.

For example:

$$(S_1 \cup S_2) \cup (T_1 \cup T_2) \simeq \bigcup(S_1, S_2, T_1, T_2) \text{ ($\cup$-Super, $\cup$-Sub)}$$

$$(S_1 \cap S_2) \cap (T_1 \cap T_2) \simeq \bigcap(S_1, S_2, T_1, T_2) \text{ ($\cap$-Sub, $\cap$-Super)}$$

$$\bigcup(T_1, T_2, T_3) \simeq \bigcup(T_2, T_3, T_1) \text{ ($\cup$-Super, $\cup$-Sub)}$$

$$\bigcap(T_1, T_2, T_3) \simeq \bigcap(T_2, T_3, T_1) \text{ ($\cap$-Sub, $\cap$-Super)}$$

$$S \cap (T_1 \cup T_2) \simeq (S \cap T_1) \cup (S \cap T_2) \text{ ($\cap$-$\cup$-Dist, $\cap$-Sub, etc.)}$$

$$S \cup (T_1 \cap T_2) \simeq (S \cup T_1) \cap (S \cup T_2) \text{ ($\cup$-Super, $\cap$-$\cup$-Dist, etc.)}$$

Simplification. The subtyping rules also allow complex type expressions to be simplified. For example:

$$\text{Where } S \preceq T,\ S \cup T \simeq T \text{ ($\cup$-Super, $\cup$-Sub)}$$

$$\text{Where } S \preceq T,\ S \cap T \simeq S \text{ ($\cap$-Super, $\cap$-Sub)}$$

$$\text{Where } S \simeq S',\ S \cup T \simeq S' \cup T \text{ ($\cup$-Super, $\cup$-Sub)}$$

$$\text{Where } S \simeq S',\ S \cap T \simeq S' \cap T \text{ ($\cap$-Sub, $\cap$-Super)}$$

2.2.4 Normalization

Before we consider extending the subtyping algorithm from section 2.1.3, we should note that equivalences like those described in the previous section complicate the task significantly. The algorithm is driven by case analysis, where the applicability of a particular case depends on the structure of the types to be compared. But if a type with some structure may be rewritten to have many other structures, isolating particular cases doesn't do much to simplify the problem.

For this reason, we'll define a normalized form for types. Algorithms operating on normalized types can make limiting assumptions that reduce complexity. The performance of such algorithms may also benefit from reduced redundancy in normalized types.

Let $|T|_\Gamma$ represent the normalized form of T in type environment Γ. We'll say that type T is *normalized under* Γ if $T = |T|_\Gamma$. Whatever the details of normalization, it must be the case that $\Gamma \vdash T \simeq |T|_\Gamma$. We also wish to make the following guarantees, producing a sort of disjunctive-normal form, *if* $|T|_\Gamma$ is a union or intersection:

- $|T|_\Gamma$ has two or more elements.
- All of $|T|_\Gamma$'s elements are normalized under Γ.
- None of $|T|_\Gamma$'s elements are a union.
- If $|T|_\Gamma$ is an intersection, none if its elements are an intersection.
- For any two elements T_1 and T_2 of $|T|_\Gamma$, it is not the case that $\Gamma \vdash T_1 \preceq T_2$.

The details of implementing a normalization achieving these goals are tedious, but, given the equivalences in the previous section, conceptually straightforward. First, distributivity and associativity are used to "lift" nested unions out of intersections and to "flatten" unions of unions or intersections of intersections. Second, each intersection (which now must contain only base types, $\top$, or $\bot$) is reduced to its minimal elements,[7] and nullary and unary intersections are converted to simpler forms. Finally, each union is similarly reduced, and nullary and unary unions are simplified.

It would be convenient if the normalized form of a type were *canonical*—that is, if every type in a particular equivalence class had the same normalized form. Testing for equivalence would then reduce to testing for equality after normalization. Unfortunately, this is difficult in the current type system, because intersections and unions can be freely permuted. We would need to arbitrarily enforce a total ordering of all types, sorting union and intersection elements appropriately. As we extend the type system with additional features, further complications will arise. Thus, we will not attempt to define a canonical normalization for types.

[7] The $\preceq$ and $\succeq$ relations are *preorders* (that is, they are reflexive and transitive). Recall that given a (nonempty) list of values and a *total order* for comparing them, a single minimum value x can be found, where *minimum* means that no value in the list precedes x; similarly, given a list of values and a preorder for comparing them, a minimal list of values $\overline{x}$ can be found, where *minimal* means that, for all x_i, no value in the original list precedes x_i. A standard, generic algorithm can thus be used to minimize intersections and unions. For the sake of determinism, we'll require that, where two types are equivalent, the leftmost type be preferred.

2.2.5 Algorithmic Subtyping

As was the case for the BASE subtyping rule in section 2.1.3, we'll need to reexpress ∪-SUB and ∩-SUPER so that the subtyping algorithm can avoid using the TRANS rule.

$$\frac{\exists i, \Gamma \vdash S \preceq T_i}{\Gamma \vdash S \preceq \bigcup \overline{T}} \ (\cup\text{-SUB}^*)$$

$$\frac{\exists i, \Gamma \vdash S_i \preceq T}{\Gamma \vdash \bigcap \overline{S} \preceq T} \ (\cap\text{-SUPER}^*)$$

Again, Let $\Gamma \vdash S \preceq: T$ represent an invocation of the subtyping algorithm for types S and T in environment Γ. It can be resolved as follows:

1. Let $|S|_\Gamma = S'$ and $|T|_\Gamma = T'$.

2. If Γ.without contains "$S' \preceq: T'$" then the result is false.

3. Otherwise, the result is true if and only if one of the corresponding tests in the following table have a true result.

		T'				
		$\top$	$\bot$	B	$\bigcup \overline{T}$	$\bigcap \overline{T}$
S'	$\top$	Top	-	-	-	-
	$\bot$	Top	Bottom	Bottom	Bottom	Bottom
	B	Top	-	Reflex, Base*	$\cup$-Sub*	$\cap$-Sub
	$\bigcup \overline{T}$	Top	-	$\cup$-Super	$\cup$-Super	$\cup$-Super
	$\bigcap \overline{T}$	Top	-	$\cap$-Super*	$\cup$-Sub*	$\cap$-Sub

The four cases in which unions or intersections are compared to other unions or intersections are of particular interest. In general, either the "super" or the "sub" rule might be applicable; however, we can show in each case that one implies the other, and so both rules need not be tested. In some cases, the implication goes both directions, and so the choice of which rule to test is arbitrary.

2.3 Arrows and Tuples

Languages with first-class functions allow functions to be treated as values—for example, using a function as input to another function, or generating functions as the result of an application. To support type-checking these languages, *arrow* and *tuple* types can be used.

Even in languages that do not have first-class functions, arrow types can be useful for analyzing *function overloading*, the declaration of multiple functions with the same name but different types. The type of an overloaded function name can be encoded

as an intersection of arrows.

2.3.1 Types

We extend the definition of types in section 2.1.1 to include the following cases:

- An *arrow type*, $S \to T$, which represents the set of functions that consume values of the first type and produce values of the second type.
- A *tuple type*, $(\overline{T})$, which represents the cross product of the sets corresponding to the given types.

2.3.2 Declarative Subtyping

We extend the definition of subtyping in section 2.1.2 with the following rules:

$$\frac{\Gamma \vdash S \preceq T}{\Gamma \vdash U \to S \preceq U \to T} \ (\to\text{-Covar})$$

$$\frac{\Gamma \vdash T \preceq S}{\Gamma \vdash S \to U \preceq T \to U} \ (\to\text{-Contravar})$$

$$\frac{\forall i, \Gamma \vdash S_i \preceq T_i}{\Gamma \vdash (\overline{S}) \preceq (\overline{T})} \ (\text{Tuple-Covar})$$

An interesting property of arrows and tuples is that they exhibit *covariance* and *contravariance*: a supertype may be determined by widening (in the case of covariance) or narrowing (in the case of contravariance) the components of a type. For

example, if Γ.extends(B, A) and Γ.extends(C, B), the subtyping rules allow a function of type $(B, B) \to B$ to be provided where the type $(C, B) \to A$ is expected.

If our language has unions and intersections, we can also include the following distribution rules (extending subtyping from section 2.2.2):

$$\Gamma \vdash (S \to T_1) \cap (S \to T_2) \preceq S \to (T_1 \cap T_2) \ (\cap\text{-}\to\text{-Dist-R})$$

$$\Gamma \vdash (S_1 \to T) \cap (S_2 \to T) \preceq (S_1 \cup S_2) \to T \ (\cap\text{-}\to\text{-Dist-L})$$

$$\Gamma \vdash (\overline{S}) \cap (\overline{T}) \preceq (S_1 \cap T_1, \ldots, S_n \cap T_n) \ (\cap\text{-Tuple-Dist})$$

$$\Gamma \vdash (S_1 \cup T_1, \ldots, S_n \cup T_n) \preceq (\overline{S}) \cup (\overline{T}) \ (\cup\text{-Tuple-Dist})$$

2.3.3 Equivalence Rules

The distribution rules for arrows, tuples, intersections, and unions in the previous section lead to corresponding equivalences.

$$(S \to T_1) \cap (S \to T_2) \ \simeq \ S \to (T_1 \cap T_2) \ (\cap\text{-}\to\text{-Dist-R}, \to\text{-Covar})$$

$$(S_1 \to T) \cap (S_2 \to T) \ \simeq \ (S_1 \cup S_2) \to T \ (\cap\text{-}\to\text{-Dist-L}, \to\text{-Contravar})$$

$$(\overline{S}) \cap (\overline{T}) \ \simeq \ (S_1 \cap T_1, \ldots, S_n \cap T_n) \ (\cap\text{-Tuple-Dist}, \text{Tuple-Covar})$$

$$(\overline{S}) \cup (\overline{T}) \ \simeq \ (S_1 \cup T_1, \ldots, S_n \cup T_n) \ (\text{Tuple-Covar}, \cup\text{-Tuple-Dist})$$

Distribution can also be used as an intermediate step to show equivalence between intersections of tuples.

$$(S_1, S_2) \cap (T_1, T_2) \simeq (S_1, T_2) \cap (T_1, S_2) \text{ } (\cap\text{-Tuple-Dist}, \cap\text{-Sub}, \text{Tuple-Covar})$$

2.3.4 Normalization

The natural strategy for normalization is to use equivance rules to translate from more complex to simpler types. However, the equivalence rules above make it difficult to determine which form of two equivalent types is "simpler." For example, consider the following equivalent intersections:

$$(A_1 \to B_1) \cap (A_2 \to B_1) \cap (A_2 \to B_2)$$

$$((A_1 \cup A_2) \to B_1) \cap (A_2 \to B_2)$$

$$(A_1 \to B_1) \cap (A_2 \to (B_1 \cap B_2))$$

The choice to, say, combine the left rather than the right side of two related arrows seems arbitrary. When further simplifications occur, it can be difficult to recognize the relationship between two equivalent types. For example, it's not obvious that the following holds where $\Gamma \vdash B \preceq A$:

$$\Gamma \vdash (A \to (B, A)) \cap (B \to (A, B)) \preceq (A \to (B, A)) \cap (B \to (B, B))$$

However, both types are equivalent (under Γ) to the following intersection, and can be derived by reducing the arrows on either their left or right side:

$$(A \to (B, A)) \cap (B \to (B, A)) \cap (B \to (A, B))$$

To normalize, then, we "expand" intersections of arrows or tuples to explicitly list all the relevant types that can be inferred from the given types. The details of this expansion are tedious, so we won't outline them here. Essentially, whenever two types in an intersection imply another, that third type is added to the intersection.

2.3.5 Algorithmic Subtyping

With the details of normalization worked out, the subtype algorithm builds trivially upon what we've already seen. We'll need a new rule combining the two rules for variance between arrows:

$$\frac{\Gamma \vdash T_1 \preceq S_1, \Gamma \vdash S_2 \preceq T_2}{\Gamma \vdash S_1 \to S_2 \preceq T_1 \to T_2} \ (\to\text{-Sub})$$

Now, reusing the algorithm outline from section 2.2.5, we simply include arrows and tuples in the case-analysis table.

		T'				
		$\top$	$\bot$	B	$S \to T$	$(\overline{T})$
S'	$\top$	Top	-	-	-	-
	$\bot$	Top	Bottom	Bottom	Bottom	Bottom
	B	Top	-	Reflex, Base*	-	-
	$S \to T$	Top	-	-	$\to$-Sub	-
	$(\overline{T})$	Top	-	-	-	Tuple-Covar

2.4 Bounded Type Variables

Type variables allow programs to abstract over types. This facilitates, for example, the definition of polymorphic functions—the function's signature can be expressed in terms of a type variable, and the type checker can produce various instantiations of that signature at application sites by providing different type arguments. To support such features, in certain contexts—such as the body of a polymorphic function—we need to be able to treat the variable itself as a type and make assertions about it that hold for all possible instantiations.

2.4.1 Types

We extend the definition of types in section 2.1.1 by including the following case:

A *variable type*, X, which is simply a name.

To limit the possible instantiations of a variable, languages often provide a mechanism to declare *variable bounds*. We'll model these declarations as functions in the type environment:

- $\Gamma.\text{upper}(X) = T$, also written $\lceil X \rceil_\Gamma = T$, defines an upper bound on X: instantiations of X are guaranteed to be subtypes of T.
- $\Gamma.\text{lower}(X) = T$, also written $\lfloor X \rfloor_\Gamma = T$, defines a lower bound on X: instantiations of X are guaranteed to be supertypes of T.

2.4.2 Well-formedness

Well-formed Instantiations. In order for reasoning about type variables to be sound, the *instantiations* of those variables must be well-formed. Informally, this means the instantiations are "in bounds."

To model variable instantiations, we'll use *substitutions*, which are mappings from variable names to their values (in this case, types). A concrete substitution is written $[\overline{X \mapsto T}]$ (that is, $[X_1 \mapsto T_1, \ldots, X_n \mapsto T_n]$); abstractly, we'll write the symbol σ. In general, a substitution *models* a logical formula, written $\sigma \models \varphi$, if the formula can be proven true under the assumption that the given variables have the given values. Similarly, a substitution may be *applied* as a transformation to a type, written σT, by replacing all instances of the given variables with the corresponding values. As a special case, σX is simply the value bound to X in σ.

Note that the type values in a substitution may contain variables, and that these

types, in general, must be interpreted in a different environment than the variables which are being instantiated. Thus, we'll need to refer to two environments: Γ, the environment for $\overline{X}$, and Γ', the environment for $\overline{T}$.

Formally:

> A substitution σ mapping from variables in environment Γ to types in environment Γ' is well-formed if and only if, for all variables in σ, $\Gamma' \vdash \sigma X \preceq \sigma \lceil X \rceil_\Gamma$ and $\Gamma' \vdash \sigma \lfloor X \rfloor_\Gamma \preceq \sigma X$.

Note that the substitution is applied to X's bounds—thus, this formalization allows for *F-bounded quantification*, or variables that are in scope within their own bounds. These sorts of recursive bounds, while introducing significant additional complexity, are quite important in object-oriented type systems.[8]

Well-formed Environments. Certain variable bounds may be unsatisfiable independent of $\overline{X}$—that is, there exists no σ such that, where Γ contains $\overline{X}$ and the well-formed environment Γ' does not, σ is a well-formed mapping of $\overline{X}$ from Γ to Γ'. For example, if $\lceil X \rceil_\Gamma = \bot$ and $\lfloor X \rfloor_\Gamma = \top$, no choice of X can satisfy these bounds.[9] We'd like to consider environments containing such bounds to be malformed.

[8] More specifically, recursive upper bounds have important use cases. It's not clear whether recursive lower bounds significantly improve expressiveness.

[9] As we'll see after defining subtyping for variables, X itself is always, by definition, a type within its declared bounds. But if X is the *only* such type, that fact is of little use in writing practical programs.

Unfortunately, checking for the existence of a satisfactory substitution is a difficult problem. We'll address it in chapter 3, which discusses type inference. For now, we'll instead settle for the following condition:

> A type environment Γ has *well-formed bounds* if there exists a well-formed environment Γ' derived by removing some variables $\overline{X}$ from Γ.upper and Γ.lower, and for all X_i, $\Gamma' \vdash \lfloor X_i \rfloor_\Gamma \preceq \lceil X_i \rceil_\Gamma$. (An environment with *no* bounds also has well-formed bounds.)

In the simple case in which X's bounds do not depend on X, this condition is equivalent to checking for a satisfactory σ. In general, however, the connection between the two is unclear. It seems likely that this check is sound but incomplete: it implies that a valid σ exists, but certain bounds for which a valid σ exists would be considered malformed. To design a correct subtyping algorithm for a particular language, the soundness property would need to be proven formally.

2.4.3 Declarative Subtyping

We extend the definition of subtyping in section 2.1.2 with the following additional inference rules:

$$\Gamma \vdash X \preceq \lceil X \rceil_\Gamma \text{ (VAR-SUPER)}$$

$$\Gamma \vdash \lfloor X \rfloor_\Gamma \preceq X \text{ (VAR-SUB)}$$

Note that if a variable's upper and lower bounds are both type T, the variable is equivalent to T.

2.4.4 Normalization

Variables can always be treated as normalized. While there are potentially equivalences between a variable and other types, in general there's not a clear ordering for simplification—if X is equivalent to Y, the choice of which to consider normalized is arbitrary.

As a performance optimization, implementations may choose to normalize a variable's bounds—that is, find a type environment with the property $|\lceil X\rceil_\Gamma|_\Gamma = \lceil X\rceil_\Gamma$ for all X in the environment (and similarly for $\lfloor X\rfloor$). We can define a function that maps Γ to a new environment Γ' where $\lceil X\rceil_{\Gamma'}$ is defined as $|\lceil X\rceil_\Gamma|_\Gamma$; but it is not necessarily the case that Γ' then has the property we're after: $|\lceil X\rceil_{\Gamma'}|_{\Gamma'} = \lceil X\rceil_{\Gamma'}$ (substituting, this is $|\lceil X\rceil_{\Gamma'}|_{\Gamma'} = |\lceil X\rceil_\Gamma|_\Gamma$). It seems likely that, for a typical normalization function, this property could be proven to hold. In other cases, given the monotonicity of normalization under a fixed environment, a normalized environment might be found by repeated application of this translation to reach a fixed point. In any case, we won't pursue such a proof here.

2.4.5 Algorithmic Subtyping

Because algorithmic subtyping cannot depend on the TRANS rule, we reexpress the variable subtyping rules as follows:

$$\frac{\Gamma \vdash \lceil X \rceil_\Gamma \preceq T}{\Gamma \vdash X \preceq T} \text{ (Var-Super*)}$$

$$\frac{\Gamma \vdash S \preceq \lfloor X \rfloor_\Gamma}{\Gamma \vdash S \preceq X} \text{ (Var-Sub*)}$$

We can extend the subtyping algorithm in section 2.1.3 by adding a line and column for variables to the case-analysis table:

		T			
		$\top$	$\bot$	B	X
S	$\top$	Top	-	-	Var-Sub*
	$\bot$	Top	Bottom	Bottom	Bottom
	B	Top	-	Reflex, Base*	Var-Sub*
	X	Top	Var-Super*	Var-Super*	Reflex, Var-Super*, Var-Sub*

Note that we must consider a number of different cases for the invocation $X \preceq: Y$: the variables may be equal, X's upper bound may be expressed in terms of Y, or Y's lower bound may be expressed in terms of X.

It's also useful to explore the interaction of variables with unions and intersections. We can extend the table in section 2.2.5 as follows:

		T'		
		X	$\bigcup \overline{T}$	$\bigcap \overline{T}$
S'	X	Reflex, Var-Super*, Var-Sub*	Var-Super*, ∪-Sub*	∩-Sub
	$\bigcup \overline{T}$	∪-Super	∪-Super	∪-Super
	$\bigcap \overline{T}$	∩-Super*, Var-Sub*	∪-Sub*	∩-Sub

In some cases, while the variable rules are applicable, they are redundant. In other cases, both the variable and the union/intersection rules must be tested.

2.5 Generic Type Constructors

The previous section facilitated programs declared abstractly in terms of types. *Type constructors* provide a mechanism for *types* declared abstractly in terms of types. For example, a $\mathsf{Map}[\![A, B]\!]$ might represent a data structure mapping values of base type A to values of base type B. More broadly, type constructors allow types to be declared abstractly in terms of any domain that is convenient: numbers, symbol or string literals, algebraic expressions, variables referring to runtime values, etc. Regardless of the domain, the following formalisms provide a framework for analyzing such types.

2.5.1 Types

We extend the definition of types in section 2.1.1 by including the following case:

> An *application type*, $K[\![\overline{a}]\!]$, where K is a type constructor (possibly defined by a program), and each a_i is a *constructor argument*.

To properly interpret constructor applications, we rely on a few relations in the type environment:

- $\Gamma.\text{params}(K) = \overline{x}$ provides a list of parameter names corresponding to the arguments $\overline{a}$; these variables may appear in some of the other environment relations involving K.

- $\Gamma.\text{constraint}(K, \varphi)$ describes a constraint which must hold for well-formed applications of K. In general, we'll allow the constraint φ to be an arbitrary logical formula. A variety of constraints might be expressible, depending on the domains of the parameters. If type parameters are allowed, we'll assume subtype assertions of the form $S \preceq T$ (the environment is implicit) can be written.

- $\Gamma.\text{subArg}(K, x) = \odot$ produces an operator representing a relation that should be used when checking that $K[\![\overline{a}]\!] \preceq K[\![\overline{c}]\!]$. The relation must be a preorder—reflexive and transitive. Syntactic equality, $=$, is the most restrictive choice for the operator, and is always a valid result. Where x is a type variable, the language might allow declarations to specify one of $\preceq$, $\succeq$, or $\simeq$ instead. Of course, the designated operator must represent a sound subtyping relationship between applications.

- $\Gamma.\text{appExtends}(K, T)$ asserts that the type $K[\![\overline{a}]\!]$ is a subtype of σT, where $\Gamma.\text{params}(K) = \overline{x}$ and $\sigma = [\overline{x \mapsto a}]$. As was the case with the similar $\Gamma.\text{extends}$ relation on base types, the details about how this relation is derived are specific

to a particular language. We have similar requirements: values of type $K[\![\overline{a}]\!]$ must actually belong to type T, and the number of types extended by a particular constructor must be finite. The relation need not be reflexive or transitive, and need not cover relationships described by Γ.subArg.

2.5.2 Well-formedness

Well-formed Applications. For a constructor application to be well-formed, it must satisfy the corresponding constraints. Formally:

> The application $K[\![\overline{a}]\!]$ is well-formed in environment Γ if and only if $\Gamma.\text{params}(K) = \overline{x}$, $\overline{a}$ and $\overline{x}$ are compatible (their arities and domains match), and for all φ such that $\Gamma.\text{constraint}(K, \varphi)$, $[\overline{x \mapsto a}] \models \varphi$.

How the assertion $[\overline{x \mapsto a}] \models \varphi$ is checked depends on the sorts of constraints that can be expressed. If φ is a subtype assertion of the form $S \preceq T$, it can be checked by testing, where $\sigma = [\overline{x \mapsto a}]$, that $\Gamma \vdash \sigma S \preceq \sigma T$.

If a type constructor is declared in a program and some of its parameters are type variables, checking the portion of the program for which these variables are in scope may rely on the assumption that all constructor applications are well-formed. To do so, we can map from φ to a pair of bounds that will appear in Γ.upper and Γ.lower. We'll write the following to denote a function that performs this extraction:

$$\text{extend}(\Gamma, \overline{X}, \varphi) = \Gamma'$$

The environment Γ' is identical to Γ, except that it defines bounds for each of $\overline{X}$. The extraction of bounds need not be complete (that is, produce bounds equivalent to φ), but it must conform to the following soundness property: for all instantiations σ of $\overline{X}$ such that $\sigma \models \varphi$, σ is well-formed mapping from Γ' to Γ. The extraction is trivial if one of S or T in the formula $S \preceq T$ is in $\overline{X}$; in general, bounds on variables appearing in S and T can be inferred following the process described in chapter 3.

Well-formed Environments. In addition to the basic constraints outlined in their definitions, the type environment relations associated with constructors must conform to certain well-formedness conditions.

If Γ.appExtends(K, T), any of $\overline{x}$ appearing in T must be compatible with the corresponding operator defined by Γ.subArg(K, x_i). Loosely speaking, "compatible" here means that if subArg allows us to map from $\overline{a}$ to $\overline{c}$, then $[\overline{x \mapsto a}]T \preceq [\overline{x \mapsto c}]T$. For example, a covariant variable must not appear in a contravariant context in T.

Additionally, appExtends is malformed if it exhibits *expansive inheritance*, as described by Kennedy and Pierce [10]. Their work demonstrates that languages with unrestricted co- and contravariant type constructors can have undecidable subtype relations; the prohibition against expansive inheritance sidesteps this undecidability problem.[10]

[10] Kennedy and Pierce's work proves decidability for a small calculus with type constructors. While we adopt their inheritance restriction to avoid undecidability in this particular area, it's certainly possible that there are similar problems lurking in other aspects of our subtyping algorithms.

Finally, the constraints expressed by $\Gamma.\text{constraint}(K, \varphi)$ must be satisfiable, and the type variable bounds extracted from φ must be well-formed.

2.5.3 Declarative Subtyping

We extend the definition of subtyping in section 2.1.2 with the following additional inference rules:

$$\frac{\Gamma.\text{params}(K) = \overline{x}, \Gamma.\text{appExtends}(K, T)}{\Gamma \vdash K[\![\overline{a}]\!] \preceq [\overline{x \mapsto a}]T} \text{ (App-Super)}$$

$$\frac{\Gamma.\text{params}(K) = \overline{x}, \forall i(\Gamma.\text{subArg}(K, x_i) = \odot \wedge \Gamma \vdash a_i \odot c_i)}{\Gamma \vdash K[\![\overline{a}]\!] \preceq K[\![\overline{c}]\!]} \text{ (App-Subarg)}$$

Note that the App-Subarg rule supports covariant and contravariant subtyping, depending on the definition of subArg. In contrast to arrows and tuples, however, we have not provided a means to distribute unions and intersections over type applications. It would be interesting to design a language with such an extension, allowing arrows and tuples to be fully modeled in terms of type constructor applications. However, while the conditions under which, for example, a generic class's type parameter can be covariant or contravariant are well-explored (and beyond the scope of this thesis), the conditions that must be satisfied to support union or intersection distribution are not.

2.5.4 Normalization

In general, it is *not* the case that, for example, $\Gamma \vdash K[\![T]\!] \simeq K[\![|T|_\Gamma]\!]$. However, if Γ.subarg allows equivalent or variant parameters, some normalization can take place:

$|K[\![\overline{a}]\!]|_\Gamma = K[\![\overline{c}]\!]$ where $\overline{c}$ is derived from $\overline{a}$ as follows (given $\Gamma.\text{params}(K) = \overline{x}$):

- If a_i is a type and $\Gamma.\text{subarg}(K, x_i)$ is one of $\simeq$, $\preceq$, or $\succeq$, $c_i = |a_i|_\Gamma$.
- Otherwise, $c_i = a_i$.

Additional cases might be added for other parameter domains.

2.5.5 Algorithmic Subtyping

Again, we can easily adjust the subtyping algorithm defined in previous sections to include type constructors by adding a few rules to the case-analysis table.

The following algorithmic rule is needed:

$$\frac{\Gamma.\text{params}(K) = \overline{x}, \Gamma.\text{appExtends}(K, U), [\overline{x \mapsto a}]U \preceq T}{\Gamma \vdash K[\![\overline{a}]\!] \preceq T} \text{ (App-Super*)}$$

The subtyping algorithm is then defined as in section 2.1.3, extending the case-analysis table as follows:

		T			
		$\top$	$\bot$	B	$K[\![\overline{a}]\!]$
S	$\top$	Top	-	-	-
	$\bot$	Top	Bottom	Bottom	Bottom
	B	Top	-	Reflex, Base*	-
	$K[\![\overline{a}]\!]$	Top	App-Super*	App-Super*	App-Subarg, App-Super*

2.6 Existential Types

Existential types allow static analysis to generalize over a number of different types that are structurally similar but vary in one or more parts. Recall that union types similarly expressed the assertion that an expression has *one of* a set of types; unlike unions, existentials allow this set to be infinite.

One common application for existential types in object-oriented languages is to support *use-site variance*. This allows a type constructor that is declared without support for variance to be treated as covariant or contravariant, dependending on the programmer's needs in a particular application. To do so, the programmer uses an existential to generalize over all choices for a type constructor argument, bound by some sub- or supertype.

2.6.1 Types

We extend the definition of types in section 2.4.1[11] to include the following case:

> An *existential type*, $\exists \overline{X}_{\varphi}.T$, which represents the union of all types T' for which there exists a valid substitution mapping T to T'.

As was the case with type constructors, existentials are constrained by an arbitrary logical formula φ which restricts the valid choices for $\overline{X}$. We'll need to interpret T in the context of these variables' bounds, so we must be able to map from φ to a set of bounds via extend$(\Gamma, \overline{X}, \varphi)$. Again, this is trivial if one of S or T in the formula $S \preceq T$ is a variable; in general, bounds on variables appearing in S and T can be inferred following the process described in chapter 3.

2.6.2 Well-formedness

An existential type $\exists \overline{X}_{\varphi}.T$ is well-formed only if T is well-formed and φ is satisfiable (and type variable bounds extracted from φ be are similarly well-formed).

2.6.3 Declarative Subtyping

We extend the definition of the subtype relation in section 2.4.3 with the following rules:

[11] We model existentials in terms of universal variables, so the variables covered in section 2.4 must be part of the language as well. Also note that existential variables are of little use without *some* form of non-atomic types, such as arrows or generic type constructors.

$$\frac{\sigma = [\overline{X \mapsto U}], \sigma \models \varphi}{\Gamma \vdash \sigma T \preceq \exists \overline{X}_{\varphi}.T} \ (\exists\text{-Sub})$$

$$\frac{\overline{Z} \text{are } \textit{fresh},\ \sigma = [\overline{X \mapsto Z}],\ \text{extend}(\Gamma, \overline{Z}, \sigma\varphi) = \Gamma',\ \Gamma' \vdash \sigma S \preceq T}{\Gamma \vdash \exists \overline{X}_{\varphi}.S \preceq T} \ (\exists\text{-Super})$$

The $\exists$-Sub rule corresponds to the traditional "close" or "pack" operation for existential types, and parallels $\cup$-Sub. It is used to identify valid instantiations of the existential.

The $\exists$-Super rule corresponds to the traditional "open" or "unpack" operation for existential types, and parallels $\cup$-Super. It makes use of an informal condition that there exist some *fresh* variables $\overline{Z}$ used to model the unknown instantiations of $\overline{X}$. This assertion implies that the names $\overline{Z}$ have not been used elsewhere by an instantiation of this rule in the derivation of type checking.[12]

2.6.4 Algorithmic Subtyping

We modify the $\exists$-Sub rule to support an arbitrary subtype as follows:

[12]Formally, this can be modeled by maintaining a list of "available" names in Γ and nondeterministically "splitting" this list whenever a rule premise uses Γ for more than one assertion; but the details are tedious and non-modular, so we avoid doing so here. In subtyping algorithms, fresh-name generation can be easily implemented using mutable state.

$$\frac{\sigma = [\overline{X \mapsto U}], \sigma \models \varphi, \Gamma \vdash S \preceq \sigma T}{\Gamma \vdash S \preceq \exists \overline{X}_{\varphi}.T} \ (\exists\text{-Sub}^*)$$

Note the presence of existentially-quantified types $\overline{U}$ in the rule premise. How are these to be generated algorithmically? Fortunately, we don't need to do so—instead, we can test the satisfiability of the entire premise using the inference techniques developed in chapter 3.

Once more, we'll use the algorithm outline developed in previous sections, and simply extend the table in section 2.4.5 to include the types defined here.

		T				
		$\top$	$\bot$	B	X	$\exists\overline{X}_{\varphi}.T$
S	$\top$	Top	-	-	Var-Sub*	$\exists$-Sub*
	$\bot$	Top	Bottom	Bottom	Bottom	Bottom
	B	Top	-	Reflex, ...	Var-Sub*	$\exists$-Sub*
	X	Top	Var-Super*	Var-Super*	Reflex, ...	Var-Super*, $\exists$-Sub*
	$\exists\overline{X}_{\varphi}.T$	Top	$\exists$-Super	$\exists$-Super	$\exists$-Super	$\exists$-Super

Chapter 3

Theory of Type Inference

3.1 Overview

In the previous chapter, we established a language of types for object-oriented languages with advanced type systems. We also defined various subtype relations for those types, which relations allow a language's static analysis to determine whether a value of some type S can be provided when a value of type T is required.

We now turn our attention to another important question for static analysis: given a program that elides certain type annotations (such as the type of a variable or the type arguments to a polymorphic function), can appropriate types be inferred that will satisfy subtype and other error checks? If so, what are those types?

Expressed another way, let's assume we've annotated the program with an *inference variable* for each elided type.[1] For example, a polymorphic function application $\mathsf{foo}(\mathsf{x}, \mathsf{y})$ has been rewritten $\mathsf{foo}[\![\alpha, \gamma]\!](\mathsf{x}, \mathsf{y})$, where the variables α and γ represent unknown types. Our goal is now to produce a substitution σ that instantiates the inference variables in a way that reflects the programmer's intent and makes the program well-typed. That substitution can then be applied to the program.

[1] An inference variable can be thought of as either an extension to the programming language—a new kind of type variable with special semantics—or as a meta-variable used to represent a (possibly-infinite) set of programs.

The problem of inferring an appropriate substitution can be further decomposed into two steps:

Constraint Reduction Produce a simple formula (a *substitution constraint*) describing the constraints on σ.

Constraint Solving Follow a straightforward process to determine a choice for σ that satisfies the substitution constraint (if one exists).

This will become more concrete in the sections that follow.

We should note that the problem of inferring a substitution that satisfies certain constraints has many applications beyond the language feature described above. We already encountered a few applications in chapter 2: checking that a variable is well-formed, extracting variable bounds from a type constructor constraint, and testing for a subtype of an existential. As we'll see below, subtype checking itself can be thought of as a special case of type inference.

The above outline of inference is very general. While the fundamentals are similar in most languages, a few important design decisions can have a major impact on inference features. These include:

Scope. The above discussion describes finding a substitution that could be applied to a program. The term "program" here may refer to a large collection of source code, a particular module, a function declaration, or even a single function application. At the broadest scope, just one substitution is inferred during static analysis; if

the scope is narrower, more substitutions are separately inferred. Inference strategies using broader scopes are called *global,* while strategies with narrower scopes are called *local.*

This distinction is important, because generally constraint solving must choose between many possible solutions. If we choose to instantiate some inference variables with only local information, we risk choosing types that lead to errors elsewhere in the program. On the other hand, local inference strategies are easier for programmers to follow and predict, which is an important design goal; and they are easier and more efficient to implement, because the number of variables is smaller and the substitution constraint is much less complex.

Completeness. An inference algorithm is *sound* if, when it produces a result, that result consists of well-typed choices for the inference variables. Clearly, soundness is essential. It is also easy to achieve—an algorithm that always fails to produce a result is sound.

Complementing soundness, an inference algorithm is *complete* if it produces a solution whenever one exists. Achieving completeness is important; it may also be quite difficult, or even impossible, assuming soundness is a prerequesite. A complete algorithm allows greater expressiveness and requires less programmer intervention. If the alternative algorithm is arbitrarily restricted, completeness may also be more predictable. On the other hand, achieving completeness may lead to more complexity, which can also lead to programmer confusion, and may negatively impact compiler

performance.

Since absolute completeness is not an essential property of an inference algorithm, it's useful to consider completeness as a relative property: one algorithm is *more complete* than another if the set of programs for which it produces a result is a proper superset of those handled by the other algorithm.

Proactive Reduction. Constraint reduction can be thought of as a process of accumulating simple constraints on inference variables. In the process of checking a program, each subtyping assertion, for example, might produce a simple constraint on a variable. If accumulation is lazy—that is, it simply accumulates a list of simple constraints—more work must be done in constraint solving. If, on the other hand, constraint reduction is proactive—merging constraints on a variable as they are produced into a single, simpler constraint—constraint solving is more straightforward. More importantly, a proactive strategy can eliminate redundancy, which may dramatically improve performance. And by immediately recognizing when the constraints on a variable are unsatisfiable, a proactive approach may lead to error messages that better isolate a problem to a local portion of the code.

3.2 Constraint Reduction

Constraint reduction is the first phase of the inference process. It is so termed because, from the start, we already have a formula describing a constraint that needs to be satisfied: we must, for example, find σ such that, where P is the program, $\sigma \models$

"P is well-formed". Our goal is to reduce this very general statement into something concrete and simple enough that constraint solving becomes a straightforward process. For example, $\sigma \models (\alpha \succeq \mathsf{String}) \wedge (\gamma \preceq \mathsf{Square} \cup \mathsf{Circle})$.

In this discussion, asserting that a program is well-formed reduces to asserting a number of subtyping relationships.[2] So our focus will be to revise subtyping such that, rather than a relation between types, it is a function producing a substitution constraint. We'll write $\Gamma \vdash S \preceq_? T | \varphi$ to mean that, under condition φ, S is a subtype of T in environment Γ. Alternately, we can just talk about the value $\Gamma \vdash S \preceq_? T$ without mentioning it by name, φ.

Substitution constraints may take the following forms:

- The literal true.
- The literal false.
- A variable lower bound $\alpha \succeq T$.[3]
- A variable upper bound $\alpha \preceq T$.

[2]Where a language performs static checks that *cannot* be expressed in terms of subtyping, a similar pattern might be followed to produce constraints from other relations on types; here, we restrict ourselves to the subtype relation.

[3]Note that we've elided Γ from the subtype constraints. For simplicity, we'll assume that the entire constraint is expressed in terms of a single, implicit type environment. In applications where that is not the case, a more general formulation might include type environments as part of the substitution constraint.

- A conjunction of constraints $\varphi \wedge \rho$.
- A disjunction of constraints $\varphi \vee \rho$.

Where neither S, T, nor Γ contain inference variables, the result of $\preceq_?$ must be equivalent to either true or false—these special cases map directly to invocations of $\preceq$. More generally, the correspondence between the two relations can be expressed as follows:

$\sigma \models (\Gamma \vdash S \preceq T)$ if and only if $\sigma \models \varphi$[4] where $\Gamma \vdash S \preceq_? T | \varphi$.

3.2.1 Subtype Reduction

Concretely, the subtype constraint reduction algorithm can be expressed simply as a modified version of the subtype algorithm described throughout chapter 2 (first outlined in section 2.1.3). Our modification closely parallels the original definition, with an additional case to handle inference variables.

1. Let $|S|_\Gamma = S'$ and $|T|_\Gamma = T'$.
2. If one of S' or T' is an inference variable, the result is defined as follows (earlier cases take precedence):
   (a) $\Gamma \vdash \alpha \preceq_? \alpha$ produces true.
   (b) $\Gamma \vdash \alpha \preceq_? \gamma$ produces $(\alpha \preceq \gamma) \wedge (\gamma \succeq \alpha)$.

[4] We haven't formally defined what it means for a substitution to model a substitution constraint, but the intent should be clear.

(c) $\Gamma \vdash \alpha \preceq_? T'$ produces $\alpha \preceq T'$.

(d) $\Gamma \vdash S' \preceq_? \gamma$ produces $\gamma \succeq S'$.

3. If Γ.without contains "$S' \preceq_? T'$" then the result is false.

4. Otherwise, a finite set of constraints, as determined by the structures of S' and T', and as outlined in a table covering our chosen domain of types, is produced. The result is the disjunction of these constraints.

   In the table, an inference rule name represents a constraint. If the corresponding rule conclusion does not match both S' and T', this is simply false; otherwise, we map the rule premise to a constraint, replacing the premise's logical assertions with substitution constraint constructors. Specifically:

   - $\Gamma \vdash U \preceq V$ becomes $\Gamma' \vdash U \preceq_? V$. The environment Γ' is produced by extending Γ with the assertion Γ'.without$(S \preceq_? T)$.
   - Simple assertions unrelated to subtyping become either true or false.
   - Logical conjunctions become constraints of the form $\varphi \wedge \rho$.
   - Logical disjunctions become constraints of the form $\varphi \vee \rho$.
   - Universal quantifiers, which must only quantify over finite domains, become constraints of the form $\varphi_1 \wedge \varphi_2 \wedge \ldots$. (If the domain is empty, this is true.)
   - Existential quantifiers, which must only quantify over finite domains, become constraints of the form $\varphi_1 \vee \varphi_2 \vee \ldots$. (If the domain is empty, this is

false.) Implicit existential quantifications (meta-variables that only occur in the premise) are handled in the same way.

As an example, consider the invocation $\Gamma \vdash (A \cup \alpha) \preceq_? (\gamma \cup B)$ where Γ.extends is empty. This can be incrementally reduced to a substitution constraint as follows:

$$(A \cup \alpha) \preceq_? (\gamma \cup B)$$

$$(A \preceq_? (\gamma \cup B)) \wedge (\alpha \preceq (\gamma \cup B)) \qquad (\cup\text{-Super})$$

$$((\gamma \succeq A) \vee (A \preceq_? B)) \wedge (\alpha \preceq (\gamma \cup B)) \qquad (\cup\text{-Sub})$$

$$((\gamma \succeq A) \vee \mathsf{false}) \wedge (\alpha \preceq (\gamma \cup B)) \qquad (\text{Base}^*)$$

3.2.2 Constraint Equivalence

As might be expected, substitution constraints can be simplified in any way that preserves the logical assertions encoded by the constraint. For example, if one term in a conjunction implies another, the second can be removed without changing the meaning of the constraint. Formally:

- $\varphi \models \rho$ if and only if, for all σ, $\sigma \models \varphi$ implies $\sigma \models \rho$.
- $\varphi \equiv \rho$ if and only if $\varphi \models \rho$ and $\rho \models \varphi$.

As a simple example, the result of $\Gamma \vdash (A \cup \alpha) \preceq_? (\gamma \cup B)$ in the previous section was:

$$((\gamma \succeq A) \vee \mathsf{false}) \wedge (\alpha \preceq (\gamma \cup B))$$

This is equivalent to:

$$(\gamma \succeq A) \wedge (\alpha \preceq (\gamma \cup B))$$

In addition to the usual rules for logical equivalence, a few important equivalences hold for subtype assertions:

$$(\gamma \preceq S) \wedge (\gamma \preceq T) \equiv \gamma \preceq S \cap T$$

$$(\gamma \succeq S) \wedge (\gamma \succeq T) \equiv \gamma \succeq S \cup T$$

$$\gamma \preceq \top \equiv \mathsf{true}$$

$$\gamma \succeq \bot \equiv \mathsf{true}$$

We can use these equivalences to define a normalization for substitution constraints. Structurally, normalized constraints are in disjunctive-normal form (a disjunction of conjunctions). In addition, every inference variable has exactly one upper bound and one lower bound in every disjunct. And any provably unsatisfiable disjuncts are removed.

Normalizing in this way has two important benefits. First, constraint solving easily reduces to finding an instantiation for a set of bounded variables. Second, it simplifies the identification of unsatisfiable constraints, which has important implications for algorithmic efficiency and error reporting.

One final equivalence is especially important for inference:

$$(\gamma \preceq U) \wedge (\gamma \succeq L) \equiv (\gamma \preceq U) \wedge (\gamma \succeq L) \wedge (\Gamma \vdash L \preceq_? U)$$

For a conjunction to be "provably unsatisfiable," we must be able to prove a contradiction from its elements. Where a variable's bounds are incompatible, this equivalence allows us to do just that—$\Gamma \vdash L \preceq_? U$ in that case will be false. Another useful application is to infer additional bounds on any variables that appear in U or L. Of course, these new bounds might, in turn, be used to infer *other* bounds. It's not clear in general whether this process will reach a fixed point.

3.2.3 Correctness

As outlined above, constraint reduction is sound and complete: σ models the original constraint if and only if it models the reduced constraint. This is because every reduction step we take is directly derived from either the subtyping algorithm (section 3.2.1) or established tautologies (section 3.2.2).

3.3 Constraint Solving

Constraint solving is the final phase of the inference process. Given a substitution constraint, the solver attempts to produce a substitution modeling the constraint.

We'll consider constraint solver inputs of the form defined in section 3.2. Further, we'll assume these have been normalized as described in section 3.2.2. So the core constraint-solving problem is to produce types that satisfy the bounds of a set of

variables. Given that capability, we can iterate through the list of conjunctions, producing a result from the first disjunct we're able to satisfy. If the algorithm fails to find a solution after iterating through the list, it reports that no solution was found.

Of course, if a variable's bounds do not contain inference variables, producing a solution is trivial: we can simply choose one of the bounds as the variable's instantiation. Even so, care should be taken in deciding *which* bound to choose, especially in algorithms that are not global. If a variable's instantiation will appear in a covariant context, for example, the lower bound is best; in a contravariant context, the upper bound is best.

Bounds that are expressed in terms of other inference variables (or recursively in terms of the variable itself) are much more difficult to handle. In such cases, choosing an instantiation for one variable restricts the set of choices available for another. In general, this is most likely an undecideable problem. A few strategies are helpful, however, in producing a solution:

- If a variable is tightly-bound—its upper and lower bounds are equivalent—we have no choice but to accept this type (or some other in the equivalence class) as the instantiation. The variable can then be eliminated from other variables' bounds.

- Checking transitive constraints—given bounds $L \preceq \alpha \preceq U$, computing $\Gamma \vdash L \preceq_? U$—can often help to strengthen the bounds on other variables (assuming this has not been done already during normalization).

- If there are variables without dependencies on others, an instantiation can be chosen (the lower bound, say), and the variable can then be eliminated from other bounds. Of course, at this point, we can't guarantee that a choice we make will be the correct one.

- In the worst case, if we're somehow able to prove the satisfiability of the variables' bounds but unable to produce a witness for that fact, we can use existential types to model the unknown (but known-to-exist) solution.

Fortunately, typical uses of a programming language are unlikely to produce the more difficult instances of this problem. While a constraint solver may not be able to guarantee completeness, it will likely be quite useful in practical situations as long as its behavior is simple and well-defined.

Chapter 4

Case Study: Type Inference in JAVA

As a case study for the type theory outlined in the previous chapters, we now consider the JAVA language. In this chapter, we'll describe the JAVA type system in terms of the theory we've developed, discuss ways in which its type inference algorithm can be improved, and examine the impact such changes would have on existing code.[1]

4.1 JAVA Type System

The specific language we'll examine is JAVA 5—the language update coinciding with the release of JAVA SE 5.0 in 2004 and specified by the 3rd edition of the *Java Language Specification* [6]. This language update introduced a number of advanced typing features, including user-defined type constructors, polymorphic methods with bounded type variables, and restricted forms of intersection types (in variable bounds) and existential types (in type constructor applications). The specification also requires some internal support for recursive types, although these are not expressible in source code.

This was a major technical addition to the language, and the JAVA language

[1] Some of this discussion was previously published in a 2008 paper presented at OOPSLA [17]. Here, the JAVA type features are framed in terms of the above theory, suggestions for constraint solving are improved, and an experimental analysis of the impact of suggested type system changes is presented.

designers, with input from the Java Community Process, spent several years carefully evaluating potential generic extensions and their technical implications. Nevertheless, the final design introduced type features that were *not* well-explored by the research literature. As a result, a number of subtle logical errors are present in the specification, and particularly in the definition of type inference. We'll examine some of these errors in section 4.2.

4.1.1 Types

Types in Java are either *primitives* or *references*; the rules for manipulating primitives are different from those for references. Because the analysis of primitives is simple and unrelated to type inference, we'll focus on reference types here, and use the term "type" to refer exclusively to references.

Java programs are organized as collections of *class* declarations, where a class describes the fields and methods associated with objects of a particular type. Classes can "extend" other classes and can be parameterized by type variables; non-hierarchical extension relationships are supported via special, restricted classes called *interfaces*. These declarations form the basis of Java types.

A type in Java is one of the following:

- The *null type*, `null`, which is similar to $\bot$ in many ways, but contains a single null value. There is no syntax for expressing this type in source code, but it is used extensively by analysis.

- A *ground parameterized class type* $C[\![\overline{T}]\!]$, which is a type constructor application with types as arguments. C is the name of a class; we'll model classes that don't have any type parameters as nullary type constructors—$\overline{T}$ in such cases is an empty list (and the brackets are then elided). One important special instance is `Object`, which is the parent class of all others, and thus acts as $\top$ in this type system.

  Class declarations can be nested within other classes. When this occurs, some type parameters for the class may be implicit from surrounding context. In our notation, the list $\overline{T}$ includes arguments for both implicit and explicit type parameters.[2]

- A *wildcard-parameterized class type* $C\langle\overline{w}\rangle$, which is an existential type wrapping a constructor application. The domain of w includes both types and *wildcards*, which represent implicitly-declared existential variables and take the form `? extends` U `super` L. Where class C has a single, unbound type parameter, the type $C\langle$`? extends` U `super` $L\rangle$ is interpreted as the existential $\exists X_{L \preceq X \preceq U}.C[\![X]\!]$. (For more complex cases, see section 4.1.3 below.)

  When we write wildcards without a lower bound, the bound `null` is implicit; similarly, the default upper bound is `Object`.[3]

[2] In the concrete syntax, a nested class's argument list may be separated into pieces like `Foo<String>.Bar<Integer>`.

[3] As specified, wildcards in JAVA must always elide at least one bound—there can be an upper bound, or a lower bound, but not both. We've generalized here by allowing both bounds at once.

Java also includes a third kind of class type, "raw types." These are class names used without type arguments, and are included for compatibility with legacy code. Fortunately, while their use can prompt certain implicit, compiler-generated casts, they are otherwise equivalent to wildcard-parameterized types. For our purposes, the raw type C is equivalent (where C has a single parameter) to $C\langle\texttt{?}\rangle$.

One final complication arises in modeling wildcards: the specified join function produces a restricted class of recursive types involving wildcards, termed "infinite types." For example, analysis might produce the type $C\langle\texttt{? extends } C\langle\texttt{? extends } C\langle\ldots\rangle\rangle\rangle$. The specification offers little guidance on how these types should be modeled or implemented [6, 15.12.2.7], and no instruction on how they relate to other types (in subtyping, for example). Some effort was made in a previous iteration of this work to properly specify these types [16], but we will make no such attempt here. Instead, we identify this lack of specification as a failing of the current type system, and suggest union types as a suitable solution.

- A *primitive array type* $p\texttt{[]}$, which is an application of a special "primitive array" type constructor.

- A *reference array type* $T\texttt{[]}$, which is an application of a special "reference array" type constructor. Unlike primitive arrays, reference arrays are covariant.

- A *type variable* X. The type environment contains upper and lower bounds for

variables. Programmers can declare upper bounds with the declaration syntax X `extends` T; no syntax supports describing lower bounds, and so only variables produced from wildcards have them. In the absense of more restrictive bounds, it is always the case that `null` $\preceq X \preceq$ `Object`.

- An *intersection type* $\bigcap \overline{T}$. Programmers can only write intersections as the upper bounds of variables: X `extends` T_1 `&` T_2. Intersections are also produced by analysis.

4.1.2 Type Environments

There is a global type environment, Γ_0, which describes the type constructors defined by all class declarations in a program. Additional distinct type environments correspond to the scope of type variables introduced by a class or method; we'll refer to these environments as Γ_C or Γ_M (M is a method name). Additionally, each top-level expression may have a distinct type environment to accomodate fresh variables used to analyze existentials.[4]

For each class declaration appearing in a program, the environment Γ_0 contains entries for the class C in the type-constructor relations described in section 2.5.1:

- $\Gamma.\text{params}(C) = \overline{X}$ where $\overline{X}$ is the list of class type parameters.

[4]The subtyping rule $\exists$-SUPER (defined in section 2.6.3) can be altered slightly to accomodate fresh variables that "already" appear in the environment Γ. This is important for JAVA because its type-checking rules also introduce fresh existential instantiations, and these variables are permitted to flow outside the scope of the subexpression in which they are introduced.

- $\Gamma.\text{constraint}(C, X_i \preceq U)$ describes an upper bound on one of X_i.
- $\Gamma.\text{subArg}(C, X_i)$ is defined as $=$ for all parameters.
- $\Gamma.\text{appExtends}(C, S)$ describes a declared supertype of the class. With the exception of `Object`, all classes have at least one entry in appExtends, and all class types ultimately extend from `Object`. The domain of S is restricted to class types.

There are also entries for the two built-in array type constructors. Each has one parameter; there are no constraints; subArg is $=$ for the primitive array constructor and $\preceq$ for the reference array constructor; and each constructor extends the two special class types `Serializable` and `Cloneable`.

4.1.3 Wildcard Capture

All wildcard-parameterized class types $C\langle\overline{w}\rangle$ represent an equivalent existential type $\exists\overline{X}_{\varphi}.C\left[\!\left[\overline{T}\right]\!\right]$. The specification defines a *wildcard capture* operation which essentially expresses this mapping.[5]

- T_i is defined as the name of a distinct variable Z_i if w_i is a wildcard, and as just w_i if w_i is a type.
- $\overline{X}$ is the list of variables introduced in the definition of $\overline{T}$.

[5] As defined in the specification, wildcard capture also expresses the generation of fresh variables that occurs whenever an existential is opened.

- φ provides an upper and lower bound for each of Z_i. Given that C has type parameters $\overline{Y}$, $w_i = \texttt{? extends } U_i \texttt{ super } L_i$, and $\sigma = [\overline{Y \mapsto T}]$, φ is a conjunction of the following for all valid i:

$$L_i \preceq Z_i \preceq (U_i \cap \sigma \lceil Y_i \rceil_{\Gamma_C})$$

Note that the upper bound of Z_i incorporates both the wildcard bound and the corresponding type parameter bound. This allows programmers to, for example, write $C\langle\texttt{?}\rangle$ without worrying about ensuring that the wildcard bounds are compatible with the declared bound.

4.1.4 Subtyping

Modeling types as described above, subtyping in Java can be expressed straightforwardly in terms of the rules described in chapter 2. The following rules are applicable:[6]

Core	Reflex, Trans, Bottom
Unions & Intersections	∩-Super, ∩-Sub
Variables	Var-Super, Var-Sub
Constructors	App-Super, App-Subarg
Existentials	∃-Super, ∃-Sub

[6]Given the restricted form of existentials in Java, the ∃-Sub rule can be simplified to only handle the case $C[\![\overline{T}]\!] \preceq C\langle\overline{w}\rangle$; in this case checking for the existence of a suitable substitution is straightforward: we simply verify that each T_i is within the bounds of (or equal to) w_i.

The algorithmic subtyping definitions corresponding to these rules can similarly be applied to JAVA subtyping.

Of course, for the algorithmic results to be correct, the types and type parameters must be well-formed, again as outlined in chapter 2. Type arguments must be within their declared bounds, and variable bounds and existential constraints (as defined by wildcard capture) must be satisfiable.

Because intersections do not distribute over any other types in this type system, no normalization is necessary. Implementations may find it useful, however, to eliminate redundant elements from an intersection, as long as this is consistent with the specification.

4.1.5 Join

Because unions are not part of the type system, the JAVA type checker occasionally must determine a common supertype of two types. We can model this with a function $\text{join}(S, T)$ which determines a common upper bound for S and T. That is:

$$\text{join}(S, T) = U \;\rightarrow\; S \preceq U \;\wedge\; T \preceq U$$

Ideally, join should produce a minimal bound, where all common supertypes of S and T are also supertypes of U. As we'll see in section 4.2.1, this isn't possible within the specified constraints of the type system. However, the function does produce reasonably tight bounds in most situations.

We won't describe the full details of join here, which involve searching for a com-

mon superclass of the two types. One interesting aspect of the function is its handling of two different parameterizations of the same class, which makes use of existentials. For example:

$$\text{join}(C[\![S]\!], C[\![T]\!]) = C\langle \texttt{? extends } \text{join}(S,T)\rangle$$

Note that the recursion in this definition may not terminate, leading to the need for special recursive wildcards, as described in section 4.1.1.

4.1.6 Type Inference

Methods in JAVA (functions bundled with an object) can declare type parameters, and invocations may either provide explicit type arguments or allow the arguments to be inferred.

Methods can also be overloaded: the expression `obj.m(x)` may refer a *set* of methods declared with name `m`. Type inference is used independently at each call site to determine type arguments for a particular method in this set; these results in turn help to determine which (if any) method should be applied. So type inference is a component of overload resolution.

The initial constraints to be solved for an inference invocation are as follows. Let M be a method with declared parameter types $\overline{T}$ and type parameters $\overline{X}$. Where the elided type arguments are represented by inference variables $\overline{\alpha}$, the signature of M can be instantiated with substitution $\sigma_\alpha = [\overline{X \mapsto \alpha}]$. Given a call site in the scope of environment Γ with argument types $\overline{S}$, inference seeks to produce an instantiation σ

for $\overline{\alpha}$ such that (for all applicable i and j):

$$\sigma \models \Gamma \vdash S_i \preceq \sigma_\alpha T_i$$

$$\sigma \models \Gamma \vdash \alpha_j \preceq \sigma_\alpha \lceil X_j \rceil_{\Gamma_M}$$

In some cases, JAVA expressions can have an *expected type* determined by the surrounding context. When this type is available, it may be used in inference to help compensate for the local nature of the algorithm. If used, the following additional constraint applies (for declared return type R and expected type V):

$$\sigma \models \Gamma \vdash \sigma_\alpha R \preceq V$$

Constraint Reduction. The specification defines a function like $\preceq_?$ to facilitate constraint reduction. Unfortunately, it diverges from the subtyping definition at times. Thus, the constraint reduction algorithm is both incomplete and unsound. It's best considered a heuristic which generally produces useful bounds; ultimately, the results of inference must be re-checked by the actual subtyping algorithm to guarantee that they are valid.

The substitution constraint produced by JAVA's constraint reduction algorithm can be thought of as two separate pieces: the first is a conjunction of bounds inferred from the method parameter types ($S_i \preceq_? \sigma_\alpha T_i$); the second is just a conjunction of the declared, already-reduced type parameter bounds ($\alpha_j \preceq \sigma_\alpha \lceil X_j \rceil_{\Gamma_M}$). Note that the

inferred bounds derived from $S_i \preceq_? \sigma_\alpha T_i$ are guaranteed not to contain any inference variables, because no inference variables appear in S_i. Also note that there are no disjuncts: restrictions in the language guarantee that these never need to be used.

Constraint Solving. Loosely speaking, the process used for constraint solving, given a set of inferred upper and lower bounds and a single declared upper bound for each variable, is as follows:

1. If some lower bounds were inferred for an inference variable, that variable's instantiation is the join of those lower bounds.

2. Otherwise, the inferred bounds are combined with the result of $\sigma_\alpha R \preceq_? V$(if V is defined in this context).

3. Finally, the instantiation for each unresolved variable is the intersection of the variable's inferred and declared *upper* bounds. (Note that this declared upper bound might include an inference variable—this is a serious problem that we'll return to in section 4.2.4.)

4.2 Suggested Improvements

The following discussion outlines a number of improvements that could be made to the JAVA type system. These suggestions build on the theory developed in this thesis to examine areas in which type inference could be made more complete and more general.

While some of the inference algorithm's current limitations arise from conscious engineering decisions, in many cases the heuristic nature of the algorithm provides a cover for unintentional specification and implementation bugs, some of which have been described in previous papers [16, 17]. Here, we'll focus on higher-level concerns.

4.2.1 Correct Join

As mentioned previously, the join function does not always produce a most specific bound. As a simple example, consider the following invocation:

$$\text{join}(C[\![\texttt{Object}]\!], C[\![A]\!])$$

The correct result in this case is $C\langle\texttt{? super } A\rangle$; the JAVA function, however, never produces wildcards with lower bounds, and will instead produce $C\langle?\rangle$.

The correct definition in other cases is more subtle. Consider a similar invocation in which the two argument types are not directly related, but share a common supertype (assume B and B' extend A):

$$\text{join}(C[\![B]\!], C[\![B']\!])$$

A tempting choice for the result (and the result chosen by the JAVA algorithm) is $C\langle\texttt{? extends } A\rangle$. However, it is equally reasonable to choose a *lower* bound for the wildcard: $C\langle\texttt{? super } B \cap B'\rangle$. Both candidates are supertypes of both $C[\![B]\!]$ and $C[\![B']\!]$; yet neither is a subtype of the other. In practice, which type is more convenient depends on how the type is used.

A joint University of Aarhus–Sun Microsystems paper introducing wildcards makes note of this ambiguity [19, 3.1], but does not mention how it can be resolved—by either producing a wildcard with *both* bounds: $C\langle \texttt{? extends } A \texttt{ super } B \cap B' \rangle$, or using a union type to represent the join: $C[\![B]\!] \cup C[\![B']\!]$. Both of these types are subtypes of our previous join candidates, and both are optimal (the first is optimal in the absence of union types). But neither is valid in JAVA, so to accommodate either approach, the language would need to be extended.

A second problem, as described previously, is that join may produce recursive types with wildcards ($C\langle \texttt{? extends } C\langle \texttt{? extends } C\langle \ldots \rangle \rangle \rangle$), and the semantics of these types is unspecified. Again, there are two alternatives. The first is to fully specify the behavior of all type operations (including subtyping, join, and inference) where recursive types are present. The second is to abandon recursive types and instead compute join using union types. This also requires adjusting the domain of all type operations, but has the advantage that algorithms involving unions are far simpler than those involving recursive types.

4.2.2 Analysis Using Full Wildcard Bounds

We saw in the discussion of wildcard capture (section 4.1.3) that the bounds of the existential variable corresponding to a wildcard implictly contain the declared bounds of the corresponding class type parameter:

$$L_i \preceq Z_i \preceq (U_i \cap \sigma \lceil Y_i \rceil_{\Gamma_C})$$

The JAVA inference algorithm is inconsistent with subtyping in its handling of wildcards: rather than reasoning about wildcards by using wildcard capture, it simply recurs on the explicit wildcard bound (U_i above). This inconsistency leads to constraints that are too restrictive, limiting the overall completeness of the inference algorithm.

The solution to this omission seems simple: just use the correct bound. However, this strategy forces us to relax simplifying assumptions the JAVA algorithm makes about its inputs. Specifically, the algorithm implicitly requires that all subtyping relationships with which it is presented can be constrainted by a conjunction of simple bounds.

Note, however, that the invocation $\Gamma \vdash S_1 \cap S_2 \preceq_? T$, where both S_1 and S_2 contain inference variables, may not conform to this scheme, because it can be satisfied by $\Gamma \vdash S_1 \preceq_? T$ *or* $\Gamma \vdash S_2 \preceq_? T$. In order to avoid the possibility that relevant information will be discarded, the algorithm must guarantee that such applications will never occur. In particular, it is designed under the assumption that a (non-inference) variable appearing in the invocation does not have bounds that refer to inference variables. Variables arising out of wildcard capture violate this assumption.

In principle, and as we've described it in chapter 3, there's no reason constraint solving should be unable to handle disjunctive constraints. It's also worth noting that many use cases in which inference based on wildcard capture would be beneficial do *not* produce disjunctions.

4.2.3 First-Class Intersection Types

As noted in the previous section, intersection types can introduce additional complexity to the inference algorithm. For this reason, their use in JAVA is extremely limited: a programmer may only express an intersection in code when it appears as the upper bound of a type variable. (Programmers may be surprised to discover that the upper bound of a wildcard *cannot* be similarly expressed with an intersection.) If we are willing to extend the inference algorithm to support disjunctive constraints, it then becomes possible to support intersections as first-class citizens in the domain of types, admitting their usage anywhere an arbitrary type can appear.

As a simple motivating example, the JAVA API includes the interfaces `Flushable` and `Closeable`, implemented by streams that support a `flush` and a `close` operation, respectively. Taking advantage of these interfaces, it might be convenient to create a thread that occasionally flushes a stream, and at some point closes it. Such a thread would need to reference a variable of type `Flushable` $\cap$ `Closeable`.

It is sometimes possible to approximate the first-class use of an intersection by introducing a type variable X with an intersection upper bound, and replacing all instances of the intersection with references to X. However, this approach is quite inconvient, and does not generalize to all use cases.

Support for first-class intersections, combined with the ability to make full use of wildcard capture during inference, provides a compelling motivation for extending the inference algorithm with support for disjunctive constraints.

4.2.4 Recursively-Bounded Type Parameters

As noted in section 4.1.6, when the Java constraint solver attempts to incorporate the declared upper bounds of the type parameters before choosing σ, it does so incorrectly and allows inference variables appearing within these bounds to leak into the calling context.

If we're interested in simply patching this specification bug, the workaroud is for inference to give up in cases that will produce such malformed results. A more useful solution is to choose the inferred lower bound, which is guaranteed to *not* contain inference variables.

If we do so, it's important to first incorporate the implicit constraint that the inferred lower bound be a subtype of the declared upper bound. This is just a special case of the equivalence rule defined in section 3.2.2:

$$(\gamma \preceq U) \wedge (\gamma \succeq L) \equiv (\gamma \preceq U) \wedge (\gamma \succeq L) \wedge (\Gamma \vdash L \preceq_? U)$$

For example, the Java API defines a class `Comparable<T>` which represents objects that can be compared (via some ordering) with objects of type `T`. Say I've defined classes C and D such that $C \preceq$ `Comparable`$[\![C]\!]$ and $D \preceq C$. If a polymorphic method declares a type parameter `T extends Comparable<T>`, and inference determines that the instantiation α has lower bound D, this lower bound is *not* a suitable choice for α: D is not a subtype of `Comparable`$[\![D]\!]$. By computing $D \preceq_?$ `Comparable`$[\![\alpha]\!]$, we can further infer that α must be equivalent to C.

4.2.5 Lower-Bounded Type Parameters

While wildcards may be bounded from either above or below, type parameters are not given this flexibility: only an upper bound is expressible. It's natural to wonder whether this inconsistency is necessary (especially given that variables produced by capture can have both upper and lower bounds). In fact, the limitation is closely tied to the type argument inference algorithm, and improvements to the algorithm would make this restriction unnecessary.

As an example use case, consider an `Option<T>` class, which can be used to represent a possibly-unknown value of type `T`. In JAVA, we can give this class a method `T unwrap(T alt)`, which returns the wrapped value, if it exists, and `alt` otherwise. This method would be more useful if `alt` could be a supertype of T:

```
<S super T> S unwrap(S alt)
```

Given the significance of lower-bounded parameters, why are they prohibited? The specification indirectly suggests that type inference cannot be easily altered handle such bounds [6, 4.5.1]. In fact, most use cases for lower-bounded parameters would be trivial to handle by simply joining the inferred and declared lower bounds. The only potential difficulty is when a declared lower bound contains inference variables; in this case, some of the constraint-solving strategies outlined earlier would help to produce useful results. (In practice, uses of lower bounds containing inference variables would probably be quite rare.)

4.2.6 Allowing `null` as a Variable Instantiation

One final change to the constraint solver is simple to implement but would likely constitute a significant practical improvement to the inference algorithm: in some cases, the best choice for an inference variable instantiation is the type `null`. The JAVA algorithm avoids such results, instead choosing an upper bound or the type `Object`.

One common use case is a factory method invocation that produces an "empty" object, such as an empty list. In such cases, there may be no information available from constraint reduction to limit the choices for α:

```
cons("foo", empty())
```

While the choice between `null` and `Object` is then essentially arbitrary, `null` is, in general, more useful. The above invocation would fail to compile given the current language's choice of `Object`, but would work fine if `null` were chosen instead, and assuming `cons` were defined as follows:

```
<T> List<T> cons(T first, List<? extends T> rest)
```

We must make an exception to this general pattern of preferring lower bounds, however: if an inference variable appears in the argument types $\overline{T}$ of the polymorphic method, and `null` is the inferred lower bound, the *upper* bound will usually be a better choice (because the inference variable most likely appears in contravariant contexts).

For example, consider the following method signature, where the `Comparator<T>` class has a method defining a total order for values of type `T`:

```
<T> Comparator<T> inverse(Comparator<? super T> c)
```

In the absense of an expected type (from the surrounding context), all invocations of `inverse` will have `null` as the inferred lower bound of the instantiation of `T`. Clearly, however, this is not what the programmer will want: a `Comparator<null>` can only compare null values! So, in this and similar cases, the best choice for instantiating `T` is the upper bound.

4.2.7 Better Use of Context

Note that the JAVA constraint-solving procedure described in section 4.1.6 uses the results of $\sigma_\alpha R \preceq_? V$ (where R is the declared return type and V is the type expected in the surrounding context) *only* if no lower bound for an inference variable is found by comparing the invocation's argument and parameter types. This limitation is somewhat arbitrary, and unnecessarily constrains the algorithm's ability to choose results that will be useful in the surrounding context. For example, the following invocation will not compile, because the method invocation produces a `Set<Integer>`, not a `Set<Number>`:

```
Set<Number> s = Collections.singleton(23);
```

Another unnecessary limitation is the restricted set of contexts in which the expected type V is defined. Assignments, variable declarations, and return statements define

expected types; no other statements or expressions do. It would be trivial to to extend this set to include conditional expressions (of the form `exp ? exp : exp`). Without making inference global, even the argument expressions of certain method invocations could have an expected type, assuming the method is not polymorphic or overloaded.

4.3 Impact on Existing Code

Enhancements to the JAVA language are generally made in a backwards-compatible fashion: the revised language is a superset of the previous version, and the behavior of previous programs is preserved. Unfortunately, changes to the current specification that affect join and type inference are almost impossible to make without rendering some programs incorrect, and changing the behavior of others.

Consider, for example, the signature for the method `java.util.Arrays.asList`:

```
static <T> List<T> asList(T... ts)
```

If this method is invoked in a context in which the expected type is unknown—as an argument to another method, for example—invariant type argument subtyping can easily cause a correct program to become incorrect with only slight modifications to the inference algorithm. That is, where the original algorithm produces $[\alpha \mapsto T]$ and the context of the invocation requires a $\texttt{List}[\![T]\!]$, an algorithm that produces a *better* but different type S will lead to an assertion that $\texttt{List}[\![S]\!] \preceq \texttt{List}[\![T]\!]$, which is false.

More troubling is the possibility that a change to join or the inference algorithm, while not invalidating a certain previously well-formed program, will change the *meaning* of that program. This is possible because overloading resolution is dependent on the types produced by type checking.

Despite these incompatibilities, existing bugs in the JAVA specification (as described in a previous paper [17]) and the shortcomings outlined in section 4.2 provide strong motivation for improving these operations.

To examine the impact of potential changes to the JAVA inference algorithm, we developed a tool which implements type checking both as specified and with an extension that addresses many of the concerns in section 4.2. Specifically, the improved checking:

- Makes use of union types
- Incorporates a sound and complete constraint reduction algorithm which can express disjunctions
- Improves the constraint solver:
  - If a parameter's declared upper bound contains an inference variable, additional bounds are derived by comparing the currently-inferred lower bound to the declared upper bound.
  - Always infers bounds from the expected type (if it is defined) before choosing variable instantiations.

  - If a parameter has no inferred lower bound and it does not appear in the method's declared arguments, the type `null` is chosen as its instantiation.

- Defines the expected type for method invocations occuring inside a conditional expression (of the form `exp ? exp : exp`)

The tool was then used to analyze the sources from four open-source JAVA projects that make significant use of type inference. Details for running this tool are provided in appendix B; summary statistics appear in the following table.

	FORTRESS	OPENJDK	DRJAVA	PLT
Lines of code	92 K	88 K	87 K	22 K
Generic methods	219	245	58	694
Annot. invocations ratio	723/3021	181/1502	9/1391	249/1154
"`super`" wildcards ratio	23/286	24/426	41/268	1784/3964
Expression types changed	593	70	9	152
Annot. removed ratio	61/361	4/85	2/7	44/160
Casts removed	3	0	2	16

Details about the sources selected for analysis appear in appendix B; note, in particular, that OPENJDK above refers only to the language tools portion of that project, not the standard JAVA APIs or the JAVA runtime implementation.

The rows of the table refer to the following:

Lines of code	The number of non-comment JAVA source lines, in thousands.
Generic methods	The number of methods with type parameters declared by the sources.
Annot. invocations ratio	Two values: the number of *annotated* ("explicit") polymorphic method invocations in the source and the *total* number of polymorphic method invocations (the rest depend on inference).
"`super`" wildcards ratio	Two values: the number of *lower-bounded* wildcards in the source and the *total* number of wildcards. Many problems in the original algorithm relate to the use of wildcards.
Expression types changed	The number of expressions (including subexpressions) with types that are not identical when comparing the results of the two algorithms. Entries counted by the next two rows require this as a prerequisite—if the type of the expression is unchanged, the algorithm changes have no impact.
Annot. removed ratio	Two values: the number of statements containing at least one annotated polymorphic method invocation in which the current algorithm requires the annotations and the improved algorithm *does not*; and the *total* number of such

	statements in which the current algorithm requires annotations.
Casts removed	The number of cast expressions which are necessary under the current algorithm but are not necessary under the improved algorithm.

While the small selection of code makes it premature to draw sweeping conclusions, it's clear from these numbers that the improved algorithm would significantly reduce the clerical burden in some programs, while others would see very little impact at all.

The negative impacts observed were minimal. In no case did the algorithm changes lead to the selection of a different overloaded method (which would allow arbitrarily-different program behavior). In only one file (five statements) did the changes lead to compilation errors.[7]

By considering the causes for annotations and casts becoming removeable, we can identify which improvements to the inference algorithm are likely to have the biggest impact on *existing* code (keeping in mind that these programs were written to satisfy the existing type checker). We noted the following major factors:

- Improved precision of join (40 annotations or casts). In particular, joining two

[7] The improved algorithm's preference for `null` rather than `Object` in the constraint solver led to the error. To be fair, the code appeared in a test that did not exercise the methods of the value with the inferred type; if it had, the programmer would have been forced to insert annotations to get useful results, and the errors would then have not occurred.

"`super`" wildcards does *not* produce a "`super`" wildcard.

- Inferring a `null` type argument rather than `Object` (34 annotations or casts). This comes up frequently for invocations that produce empty lists and similar objects.

- Always using the expected type (35 annotations). The JAVA inference algorithm ignores the expected type if lower bounds can be inferred from the argument types, and so can infer arguments that are incompatible with the expected type.

Note that these are all fairly simple problems with straightforward solutions—many of the subtle issues described in previous sections, while important for correctness, have little impact on these concrete examples.

In summary, despite the theoretical concerns, it seems possible to develop an improvement to the existing Java type inference algorithm that would significantly improve its handling of some programs while requiring minimal transitional problems for legacy code. Of course, such a transition would require some tool support to allow programmers to *guarantee* that their program behavior would not change, but the effort required to migrate using such a tool would probably be small.

Chapter 5

Conclusion

This thesis has provided a formal framework for discussing types, subtyping, and type inference in the context of object-oriented languages with advanced typing features. It has also demonstrated how this formal framework can be applied to the JAVA language, and showed how principled improvements to the language's type system can be applied to produce a more expressive language without negative practical consequences.

Some specific, unique contributions of this work include the following:

- Exploring the interaction of unions and intersections with other typing features.
- Defining a type inference algorithm as a generalization of subtyping, produced by a straightforward transformation applied to the subtyping algorithm.
- Identifying useful applications of type inference to a variety of type-checking tasks, including checking constraint well-formedness and existential subtyping.
- Formally expressing JAVA wildcards as existential types.
- Defining and implementing a concrete JAVA inference algorithm that improves on the specification.
- Performing a practical analysis of the impact of inference algorithm changes on

existing code, noting that, without introducing significant backwards-incompatibility problems for these programs, we've managed to significantly reduce the need for annotated method invocations.

The formal framework defined here was intended to be general enough to represent the core typing features of languages like SCALA, FORTRESS, and X10. Future work might explore this connection in a concrete manner, as we did with JAVA, looking for opportunities to improve both the framework and the languages themselves. In particular, it is hoped that this work will help in the design of type inference specifications for these and similar languages.

There is also ample opportunity to strengthen the foundations of our formal framework. For example:

- We've mentioned throughout this thesis important assumptions that ought to be proven, such as the equivalence of declarative and algorithmic subtyping.
- The intuitive connection between types and sets could be explored formally by developing a denotational semantics for these types.
- There are a number of important results in type theory concerning the satisfiability of type inference; these should be related to our inference approach, in order to better understand the circumstances under which constraint solving is decidable.

Finally, we've limited our scope in this thesis to *designing* type inference. *Imple-*

mentation is a separate, complex problem. Particularly important is the ability for analysis to scale with program size.

5.1 Related Work

5.1.1 General

All of the typing features described in this thesis have a long history in the field of type theory; constraint-producing type-checkers are also well-established. Pierce's *Types and Programming Languages* [14] is a very good general-purpose reference on type theory, and covers most of these topics. It also has an extensive bibliography referencing the most important works in the field, which we won't reproduce here.

All of the major languages mentioned in this thesis have official specifications (some of them only drafts) which address type inference with varying degrees of concreteness. These include JAVA [6], C# [21], SCALA [11], FORTRESS [2], and X10 [15]. Many of the ideas in this thesis were directly inspired by the JAVA and FORTRESS specifications.

Barbanera, Dezani-Ciancaglini, and Liguoro [3] explored subtyping in the presense of both unions and intersections. In particular, their paper presents distribution rules for intersections of unions and intersections of arrows.

Algorithms for local type inference in languages with subtyping and bounded quantification were first explored by Cardelli [5] and later Pierce and Turner [12, 13]. Pierce and Turner noted the difficulty of performing inference for type parameters

with interdependent bounds, and did not handle these instances [12].

Kennedy and Pierce [10] demonstrated the undecidability of subtyping algorithms (and, by extension, subtype inference algorithms) for some object-oriented type systems that can express contravariance. Fortunately, their work also suggests a straightforward well-formedness limitation to the class extension graph which restores decidability to their simplified calculus.

Union types are explored in the context of object-oriented languages by Igarashi and Nagira [9]. Their work provides an interpretation of inheritance for union types, a problem we have not explored here. Their approach is reminiscent of structural subtyping; a more nominal approach could also be developed.

5.1.2 JAVA

Many of the advanced typing features in JAVA, including type variables and parameterized types, accompanied by inference, were adopted from the GJ language [4], an extension to JAVA designed to support generic programming.

Wildcards arose out of research to extend GJ and similar languages with covariant and contravariant subtyping. Thorup and Torgersen [18] initially proposed what has become known as use-site covariance—allowing programmers to specific when a parameterized type is instantiated that a particular type parameter should be covariant. Igarashi and Viroli [8] extended this notion to include contravariance and established a connection to bounded existential types. In contrast to GJ, their work required

support for variables with lower bounds. A joint project between the University of Aarhus and Sun Microsystems [19] extended these ideas and merged them with the rest of the JAVA language, describing in particular how wildcards affect type inference. Wildcard capture was first presented in this paper.

The JAVA specification (3rd edition) [6] enhanced this prior work in a number of ways. Wildcard capture was refined to produce variables whose bounds include both those of the wildcard and those of the corresponding type parameter. This enhancement has a number of interesting side effects: first, intersection types are necessary to express the upper bound of some variables; second, a variable may have *both* an upper and a lower bound; and third, such variables may appear in their own upper bounds. Perhaps spurred by the requirement for intersections produced by capture, the language was also extended to allow intersection types as the bounds of declared type variables. In addition, the join operation was allowed to produce recursive types, an approach that was avoided in the Aarhus–Sun paper due to its complexity [19].

Torgersen, Ernst, and Hansen [20] complemented the specification with a formal discussion of wildcards as implemented in JAVA, and presented a core calculus extending FEATHERWEIGHT GJ [7] with wildcards. Their calculus, for the sake of generality, allows arbitrary combinations of upper and lower bounds on both declared type variables and wildcards. Their paper does not, however, discuss how such generality might affect the full JAVA language, and type inference in particular.

Appendix A

Symbol Naming Conventions

Throughout this document, variables with certain names are used to represent values in a certain domain, as outlined below. The "variable" column lists the preferred variable name; "alternates" lists additional names that may be used when multiple names are needed.

Variable	Alternates	Domain	See Also
i	j, k	Indices into a list	
n	m	Maximum indices into a list	
T	L, P, Q, R, S, U, V	Types	Section 2.1.1
B	A	Base types	Section 2.1.1
Γ		Type environments	Section 2.1.2
X	Y, Z	Type variables	Section 2.4
σ		Substitutions	Section 2.4
K		Type constructors	Section 2.5.1
x	y, z	Type constructor parameters	Section 2.5.1
a	c	Type constructor arguments	Section 2.5.1
φ	ρ	Logical formulas	Section 2.5.1

Variable	Alternates	Domain	See Also
α	γ	Inference variables	Section 3.1
P		Programs	Section 3.2
C	D	Class names	Section 4.1.1
w		Wildcards or types	Section 4.1.1
p		Primitive types	Section 4.1.1
M		Method names	Section 4.1.1

Appendix B

Code Analysis with DynamicJava

DynamicJava is an open-source interactive interpreter for Java. It was adopted by the DrJava IDE as the evaluation engine for its "interactions pane" feature [1]. The source has since been modified extensively as a component of DrJava, in particular to support generic type checking and other features of Java 5 (the original authors do not actively maintain the project).

For the purposes of this thesis, DynamicJava was extended to support static analyis in a batch mode, rather than the usual checking that occurs immediately before interpreting a snippet of code. In addition to simple error checking (mimicking a compiler front-end), the batch mode allows different type systems and other configurable options to be used to analyze the same program. A report comparing the results (two annotated abstract syntax trees) is then generated.

B.1 Running the Batch Processor

The DynamicJava sources are available at `http://drjava.org` (currently under the "Components" link). To invoke the batch processor, run the following class with a list of source file or directory names:

```
edu.rice.cs.dynamicjava.sourcechecker.SourceChecker
```

B.1.1 Options

By default, this checks the given sources with each of a hard-coded set of option configurations and compares their results. A single configuration can be invoked by itself using a "`-opt` *`name`*" argument. The following configurations were developed:

`jls` An implementation of error checking per the *Java Language Specification* [6] (simplified to ignore some classes of errors, such as unassigned variables); modified to mimic Sun's JAVAC compiler where difference from the specification became apparent.

`ext` Error checking with an "extended" type system, following the improvements described in chapter 4.

`jls-inferred` The `jls` configuration, modified to ignore all explicit type argument annotations on constructor and method invocations.

`ext-inferred` The `ext` configuration, similarly modified to ignore all explicit type argument annotations.

Custom configurations can be defined by modifying the source code.

When sources need to be checked against existing class libraries, the "`-cp` *`path`*" argument can be used to specify a class path, as in JAVAC.

B.1.2 Output

Upon invocation, the batch processor uses progress indicators to provide feedback during analysis.[1] If any errors are found, they are then printed, along with the corresponding source location.

Unless a "`-opt`" argument was given, this process is repeated for each option configuration. Once the analysis completes, a report is then generated, comparing the annotated abstract syntax trees produced by each configuration. Where differences are encountered that follow a recognized pattern (the type of an expression doesn't match, for example), they are accumulated into a list of statistics; if the pattern isn't recognized, a message is logged.

The printed list of statistics includes counts for (and, if the "`-verbose`" argument is used, a list of) the following:

Errors
: Statements that contain errors are recorded and not processed further. The report distinguishes between errors that are common to both configurations, those that only occur under the first ("left") configuration, and those that only occur under the second ("right").

  Ideally, analyzing code that cleanly compiles under JAVAC should produce no errors under the `jls` configuration. However, limi-

[1] As DynamicJava was designed to handle only small code snippets at a time, processing significant source trees may require an embarssingly large amount of time and memory.

tations in the `jls` implementation can lead to some unexpected errors. In particular, `jls` differs from the language specification or JAVAC in the following ways:

- Unchecked conversion from a raw to a parameterized class type is not supported (with the exception of unchecked casts).
- Name resolution and identification of certain fields as static constants sometimes fails.
- Resolution of overloaded methods with variable-length arguments follows the specification in unusual cases in which JAVAC fails to do so.
- In certain type inference cases involving type parameters that only appear in a method's return type, JAVAC uses an unclear process to infer a better type than the specified result `Object`.

Errors that occur under `ext` but not `jls` highlight use cases in which the modified type system fails to handle legal existing code. (As a known bug, the implementation sometimes fails to identify a member of a union type.)

Finally, errors produced by `jls-inferred` and `ext-inferred` can be used to identify cases in which explict type arguments are unnecessary (for example, if `ext-inferred` has five fewer errors than

`jls-inferred`, there are five cases in which the improved inference algorithm eliminated the need for explicit type arguments).

Feature usage — This includes counts for declarations of polymorphic methods, invocations of polymorphic methods (either with or without explicit type arguments), and wildcard instances (unbound, upper-bound, or lower-bound).

Mismatched types — Whenever the type inferred for an expression differs under two configurations, that difference is recorded. Often, the difference has no visible effect in the given sources, because the context of the expression does not distinguish between the two types.

Note that if the difference in types leads to a different method being selected by overload resolution, that fact will be logged separately in the output.

Extra casts — If a cast expression is necessary under the first configuration but not under the second (that is, a difference in the expression's type makes it a downcast under the first configuration and an upcast under the second), it is recorded as a "left extra cast." The difference in the opposite direction is recorded as a "right extra cast."

B.2 Analyzed Code Samples

Section 4.3 discusses the results of running the DynamicJava batch processor on the sources of a handful of projects. Instructions for reproducing this analysis are outlined below.

Fortress Version 4337, available at `http://projectfortress.sun.com`. Because Fortress is implemented in both Java and Scala, the compiled Scala classes in `ProjectFortress/build` must be added to the class path for analysis. Other class path dependencies include `jar` files appearing in `ProjectFortress/third_party` and the JDK library `tools.jar`, distributed with Java.

A problem encountered in analyzing the Fortress sources is that the amount of code and the unusually large number of classes are too much for a system with 4 GB of memory to handle in a reasonable amount of time. As a result, the generated sources (packages `com.sun.fortress.nodes` and `com.sun.fortress.parser`) were excluded from analysis, and the additional sources were split into tractable pieces. This works because `ProjectFortress/build` already includes class files for all classes excluded from a particular run.

OpenJDK Version 7 build b78, available at `http://openjdk.java.net`. Only the `langtools` source tree was analyzed (found in `langtools/src/share/classes`). No external dependencies need to be listed in a class path.

DrJava	Revision 5211, available at `http://drjava.org`. Some source files must be generated before analysis via `ant generate-source`. The JDK library `tools.jar`, distributed with Java, must appear on the class path, in addition to the `jar` files located in the `lib` directory.
PLT	Revision 5175, available at `http://drjava.org`. No external dependencies need to be listed in a class path.

Bibliography

[1] Eric Allen, Robert Cartwright, & Brian Stoler. *DrJava: A Lightweight Pedagogic Environment for Java.* SIGCSE, 2002.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, Jr., & Sam Tobin-Hochstadt. *The Fortress Language Specification, Version 1.0.* http://projectfortress.sun.com, 2008.

[3] Franco Barbanera, Mariangiola Dezani-Ciancaglini, & Ugo de'Liguoro. *Intersection and Union Types: Syntax and Semantics.* Information and Computation, vol. 119, no. 2, 1995.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, & Philip Wadler. *Making the Future Safe for the Past: Adding Genericity to the Java Programming Language.* OOPSLA, 1998.

[5] Luca Cardelli. *An Implementation of* $F_{\leq}$. Research report 97, DEC Systems Research Center, 1993.

[6] James Gosling, Bill Joy, Guy Steele, & Gilad Bracha. *The Java Language Specification, Third Edition.* Addison Wesley, 2005.

[7] Atsushi Igarashi, Benjamin C. Pierce, & Philip Wadler. *Featherweight Java: A Minimal Core Calculus for Java and GJ.* OOPSLA, 1999.

[8] Atsushi Igarashi & Mirko Viroli. *On Variance-Based Subtyping for Parametric Types.* ECOOP, 2002.

[9] Atsushi Igarashi & Hideshi Nagira. *Union Types for Object-Oriented Programming.* Journal of Object Technology, vol. 6, no. 2, February 2007.

[10] Andrew J. Kennedy & Benjamin C. Pierce. *On Decidability of Nominal Subtyping with Variance.* FOOL/WOOD, 2007.

[11] Martin Odersky. *The Scala Language Specification, Version 2.7.* http://www.scala-lang.org, 2009.

[12] Benjamin C. Pierce & David N. Turner. *Local Type Argument Synthesis with Bounded Quantification.* Technical report TR495, Indiana University, 1997.

[13] Benjamin C. Pierce & David N. Turner. *Local Type Inference.* POPL, 1998.

[14] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[15] Vijay Saraswat. *Report on the Programming Language X10, Version 2.0.3.* http://x10.codehaus.org, 2010.

[16] Daniel Smith. *Completing the Java Type System.* Master's thesis, Rice University, 2007.

[17] Daniel Smith & Robert Cartwright. *Java Type Inference Is Broken: Can We Fix It?* OOPSLA, 2008.

[18] Kresten Krab Thorup & Mads Torgersen. *Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes.* Lecture Notes in Computer Science, 1999.

[19] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, & Neal Gafter. *Adding Wildcards to the Java Programming Language.* SAC, 2004.

[20] Mads Torgersen, Erik Ernst, & Christian Plesner Hansen. *Wild FJ.* FOOL, 2005.

[21] *C# Language Specification, Version 3.0.* http://msdn.microsoft.com/vcsharp, 2007.