TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

Informatik-Bericht Nr. 2010-02

An Abstract State Machine Model for the Generic Java Type System

Daniel Grunwald[1], Malte Lochau[1], Egon Börger[2], and Ursula Goltz[1]

November 8, 2010

[1]Institut für Programmierung und Reaktive Systeme,TU Braunschweig
[2]Dipartimento di Informatica, Università di Pisa

Abstract

When discussing properties such as type safety for the Java language, it is necessary to have formal semantics. The complex type system involving parameterized types with wildcards, combined with the fact that some constructs are underspecified in the current version of the language specification, is making a complete formalization difficult.

In order to specify the semantics, we make use of an existing solution: an Abstract State Machine model for Java. We describe the existing model, which does not include support for generics, and extend it with the generic type system. For this purpose, we look at existing type theoretic approaches to Java wildcards. Moreover, we redefine the semantics of method invocations to handle both generic methods with type inference and other language changes such as auto-boxing.

We also present an executable version of this model which allows interactively exploring the Java semantics. Finally, we discuss the possibility of using the model to prove type safety for Java in the future.

Keywords Type Theory, Type Safety, Java Generics, Java Wildcards, Abstract State Machines

Zusammenfassung

Beim Betrachten von Eigenschaften wie Typsicherheit für die Sprache Java ist es nötig, eine formale Semantik zu haben. Das komplexe Typsystem, welches parametrisierte Typen mit Wildcards enthält, sowie die Tatsache, dass einige Konstrukte in der aktuellen Sprachversion unterspezifiziert sind, erschwert eine vollständige Formalisierung.

Um die Semantik anzugeben, bauen wir auf eine vorhandene Lösung auf: ein Abstract State Machine-Modell für Java. Wir beschreiben das bestehende Modell, welches keine Unterstützung für Generics enthält, und erweitern es um das generische Typsystem. Dazu betrachten wir vorhandene typtheoretische Ansätze zu Java Wildcards. Außerdem definieren wir die Semantik von Methodenaufrufen neu, um generische Methoden mit Typinferenz sowie anderen Sprachänderungen wie Auto-Boxing zu modellieren. Wir präsentieren auch eine ausführbare Version dieses Modells, die es erlaubt, die Java-Semantik interaktiv zu erkunden. Schließlich diskutieren wir die Möglichkeit, mit Hilfe des Modells einen Typsicherheitsbeweis für Java zu erstellen.

Schlüsselwörter Typtheorie, Typsicherheit, Java Generics, Java Wildcards, Abstract State Machines

Foreword

The contents of this Technical Report were initially published in Daniel Grunwald's Bachelor's Thesis prepared at the Institute for Programming and Reactive Systems (IPS), TU Braunschweig. The Thesis was completed in August, 2010.

The idea for the subject of this Thesis originates from preceding cooperative discussions of Egon Börger (Dipartimento di Informatica, Università di Pisa), and Ursula Goltz and Malte Lochau (IPS) during a meeting at the ETH Zurich in January, 2010. The intention was to investigate which major adjustments are required for updating the first edition of the JBook [SSB01], authored by Robert Stärk, Joachim Schmid, and Egon Börger in 2001, concerning the new features of the Java language introduced since version 5.0. Namely, the ASM models for the Java Interpreter, the Java Virtual Machine, and the corresponding Java Compiler, as well as the main proofs of the book concerning type safety and compiler correctness statements were under consideration. Apparently, the memory model and the new generic type system constitute the major innovations of Java 5.0, where the former mainly deals with compiler correctness issues in the context of multi threading capabilities, against what the latter (being subject of this work) seriously affects type safety properties of Java. One major goal when enhancing the JBook definitions was to, as far as possible, conservatively refine existing ASM models, and to give reconcilable redefinitions, otherwise. The main concept followed by Mr. Grunwald in his work was elaborated in a very fruitful and intensive further meeting with Egon Börger, Ursula Goltz, and Malte Lochau in Zurich in May, 2010. We further thank Daniel Smith for helpful hints concerning subtle details of Java's type inference as described in his Master's Thesis from November, 2007. As a result, we obtained, to the best of our knowledge, one of the first comprehensive, integrated models based on the well-defined ASM formalism, that captures all essential mechanisms of the Java type system, especially including the subtleties of the wildcard construct. During the preparation of this work, we received rich insights into technical details, difficult corner cases, as well as bugs and open problems of the new Java type system.

In addition to the theoretical definitions contained in this Report, Mr. Grunwald developed as a further part of his Thesis a prototypical implementation of the model by enhancing the original model of the JBook using the ASMGofer Framework.

Contents

List of Figures

List of Abbreviations

ASM	Abstract State Machine
AST	Abstract Syntax Tree
GUI	Graphical User Interface
Jbook	Java and the Java Virtual Machine [SSB01]
JLS	Java Language Specification [GJSB05]
JVM	Java Virtual Machine [LY99]
$Java_I$	Imperative Core of Java
$Java_C$	Procedural Extension
$Java_O$	Object-Oriented Extension
$Java_E$	Exception-Handling Extension
$Java_T$	Concurrent Exception
lub	least upper bound

1 Introduction

1.1 Motivation

In version 1.5, the Java programming language [GJSB05] underwent major changes. Generic types and methods were introduced, allowing the use of parameterized polymorphism.

To talk about properties such as type safety, we need a formalism for describing the language semantics. The book „Java and the Java Virtual Machine: Definition, Verification, Validation“ [SSB01] uses Abstract State Machines to develop a formal model of the semantics of the Java language. This model is then used to prove the type safety of the Java language. Moreover, the book presents a formal model for JVM byte code, again using Abstract State Machines, and shows a compilation schema from Java to byte code to be correct.

However, the book is referring to the second edition of the Java specification, which did not include generics. The addition of generics is a pervasive change to the type system, requiring changes to the semantics of several other language features. The majority of these changes affect the static semantics, such as the subtyping relation or overload resolution for method calls.

The goal of this Bachelor's Thesis is to introduce a formal model for the semantics of Java with generics and wildcards. In the future, this could be used to prove (or disprove) the type safety of Java with generics.

1.2 Contribution

This thesis defines a type system for Java with generics, and modifies the ASM model given in [SSB01] for use with this type system. In particular the logic related to resolution of method calls had to be revised in order to support the new language features.

We take a look at various approaches to formalize the Java 1.5 type system, and decide on using the join-based type system from [Smi07]. This type system handles problematic aspects of the Java language like infinite types, whose properties are left unspecified by the JLS.

Finally, we provide an implementation of our model. It is based on the ASM implementation included with the Jbook and is written in the AsmGofer programming language.

1.3 Structure

In the next chapter, we describe Abstract State Machines and their relation to the Java programming language. We also take a look at the Generics feature introduced in Java 1.5 and point out a few problems with it.

Chapter 3 describe the generic type system in detail, and define operations and relations (such as subtyping) on the type system.

In chapter 4, we discuss the extensions to the Abstract State Machine Model for Java that are necessary to support both the compile-time and run-time semantics of generics.

Then, in chapter 5, we give an overview of our implementation of Java Generics in the AsmGofer programming language. This is an executable version of the model described before.

In the last chapter, we conclude.

2 Fundamentals

This chapter describes the previous work on which this Bachelor's Thesis builds upon: Abstract State Machines, the „Jbook“, and the additions to the type system of the Java programming language in its version 1.5.

2.1 Abstract State Machines

Abstract State Machines are described in [BS03]:

> The notion of ASMs captures some fundamental operational concepts of computing in a notation which is familiar from programming practice and mathematical standards. In fact it is correct to view basic ASMs as „pseudo-code over abstract data“, since their (simple) semantics supports this intuitive understanding by a precise notion of a tunable abstract state and state transitions [...].

Abstract State Machines consist of a finite number of *transition rules* of the form

if *Condition* **then** *Updates*

[BS03, §2.2.2]. The condition can be any Boolean predicate, which can access the current state. The updates are rules for transforming the current state into a new state.

The state in Abstract State Machines is modeled using so-called *dynamic functions*. Updates can define or change the values of these functions by assigning to locations within the function:

$$f(t_1, \ldots, t_n) := t$$

If several updates are executed in a single computation step, the changes take effect simultaneously: all changes take effect only with the beginning of the next step. However, such update sets need to be consistent – in a single step, all assignments to a location must assign the same value.

This *synchronous parallelism* can also be used to operate on several elements in a set at the same time:

forall x **with** φ **do** R

This executes the rule R in parallel on each x that satisfies the Boolean condition φ.

Abstract State Machines have different kinds of functions: [BS03, §2.2.3]

- *static* functions depend only on their parameters and are given as part of the initial state. They can be specified in different ways depending on how the ASM is intended to be used.
- *dynamic* functions can be updated by the ASM. Dynamic 0-ary functions can be used to model variables, whereas dynamic functions with parameter can be thought of as arrays or hash tables.
- *derived* functions can depend on the current state, but cannot be directly updated by the ASM. They are intended to be used as auxiliary function.

Please refer to [BS03] for a detailed definition of the semantics of ASMs.

ASMs can be composed to form a new machine that executes both machines in parallel. This allows adding conservative extensions to an existing machine [BÖ3].

As the next section will show, this approach can be used to split up the semantics of a programming language into manageable chunks. Starting from the language core, other language constructs can be modeled as conservative extensions to the previous ASMs.

2.2 Java and the Java Virtual Machine

The book „Java and the Java Virtual Machine: Definition, Verification, Validation“ [SSB01] describes the Java language, by specifying its semantics in terms of an Abstract State Machine. The Jbook refers to the second edition of the Java specification, but does not include the new languages features such as inner classes. So the language being modeled

This ASM model of Java is then used to prove type safety by showing that, for all runs of the machine, every value calculated by an expression is a subtype of the static type of the expression. Moreover, the type safety proof shows several other properties, such as the impossibility to read a variable before it is assigned a value.

In the second part, the Jbook deals with the semantics of JVM byte code execution and verification. It then specifies a compilation function that transforms a Java parse tree into byte code. Finally, a proof of compiler correctness shows that all runs of the compiled byte code on the JVM Abstract State Machine are equivalent to runs of the Java code on the Java interpreter abstract state machine.

For the description of the Java semantics, the language is decomposed into five sublanguages. Each sublanguage is a conservative extension of the preceding sublanguage, adding additional rules and state informations to the Abstract State Machine and refining some functions [BB08].

- $Java_I$: Imperative core
- $Java_C$: Procedural (static methods)
- $Java_O$: Object-oriented constructs.
- $Java_E$: Exception Handling
- $Java_T$: Concurrency

For each sublanguage, the book first introduces the relevant Java syntax, then

discusses the constraints on valid programs. For example, the inheritance relation must be acyclic. Finally it defines the dynamic semantics using ASM rules.

2.2.1 Imperative Core

$Java_I$ defines the semantics of the imperative core of the Java language. This is a while-language with eight primitive Java types (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`). The $Java_I$ sublanguage does not contain classes or methods. The interpretation of expressions and statements is defined using ASM rules.

The interpreter works by replacing nodes of the parse tree by values. An instruction pointer stored in the nullary ASM function *pos* specifies the node that should be processed next. To determine the next step, the machine either considers the node at *pos*, or, if *pos* points to an already evaluated value, its parent node at $up(pos)$.

As an example, here is the semantics of the conditional ternary operator (`a?b:c`):

$$^{\alpha}exp_0 \; ? \; ^{\beta}exp_1 : \; ^{\gamma}exp_2 \rightarrow pos := \alpha$$

In this notation, α, β, and γ are position markers, each referring to one node of the parse tree. This rule says that the conditional operator evaluates the condition (exp_0) first.

Once the condition has been evaluated, the machine has replaced it with the evaluated value:

$$^{\blacktriangleright}val \; ? \; ^{\beta}exp_1 : \; ^{\gamma}exp_2 \rightarrow \textbf{if } val \textbf{ then } pos := \beta \textbf{ else } pos := \gamma$$

As *pos* now points to a value, the machine now considers the parent expression $up(pos)$. In this notation, $\blacktriangleright$ represents the active position *pos*. Depending on the evaluated value *val*, the machine will evaluate exp_1 or exp_2:

$$^{\alpha}True \; ? \; ^{\blacktriangleright}val : \; ^{\gamma}exp_2 \rightarrow yieldUp(val)$$
$$^{\alpha}False \; ? \; ^{\beta}exp_1 : \; ^{\blacktriangleright}val \rightarrow yieldUp(val)$$

These two rules use the *yieldUp* helper rule to pass up the value returned by the expression. This is done by setting *pos* one level up (to $up(pos)$) and then replacing that node with *val*.

A statement that has been successfully executed is replaced with the special value *Norm*, indicating normal completion.

Both statements and expressions can also be replaced with special values called abruptions, which indicate executions that end abnormally. These are used to model the control flow. In $Java_I$, abruptions are used for the `break` and `continue` statements. Abruptions are propagated upwards in the parse tree until they are handled. In case of `break` and `continue` abruptions, this happens when the abruptions reaches a loop with the matching label.

The following sublanguages will extend the use of abruptions to model the `return` statement and exception handling.

2.2.2 Procedural Extensions

The $Java_C$ sublanguage introduces classes, interfaces, and methods. However, classes and interfaces are used only as modules containing static methods and variables (or in the case of interfaces: only for constants).

Method calls are implemented in the Abstract State Machine by pushing the ASM variables used by $Java_I$ onto a stack. Likewise, when execution of a method has finished and the parse tree of the body of the method has been replaced by a return value, the stack frame is removed from the stack and the execution context of the caller is restored, except that the invocation of the method is replaced by the return value (the value associated with the `return` abruption).

The Jbook does not formally specify how the overload of a method being called is chosen by means of ASM rules. Instead, overload resolution is assumed to occur in an elaboration phase that annotates the parse tree before the Abstract State Machine starts to execute. This allows the rules of the ASM to assume that each method call is already annotated with the correct overload.

2.2.3 Object Orientation

In the $Java_O$ sublanguage, the book introduces objects. For this purpose, the set of types, which only contained primitive types in the previous sublanguages, is extended with classes, interfaces and arrays. The subtyping relation is extended to include the relations between these new types.

The semantics of object creation, instance field access and instance method calls are introduced.

To model the heap, the Jbook uses the following definition:

$$\textbf{type } Val = ...|Ref|null$$
$$\textbf{data } Heap = Object(Class, Map(Class/Field, Val))$$
$$heap : Ref \rightarrow Heap$$

This definition uses *heap* as a dynamic function which contains all objects created on the Java heap. Every object stores a mapping from the instance fields to the current values. Values can be either primitive values, references to heap objects, or the special null reference.

On instance method calls, the method being called is searched in the following way: First, the compile-time behavior of Java is modeled by assuming that overload resolution has already happened and that the syntax tree is annotated with the virtual method being called. The overload resolution algorithm is given in pseudo-code.

At runtime, when evaluating a virtual method call, the ASM uses the *Class* stored with the heap object to find the actual method being called. For this purpose, runtime method search is defined using the auxiliary *lookup* function, again, defined using pseudo-code.

2.2.4 Exception Handling

$Java_E$ adds exception handling by adding the `throw` and `try-catch-finally` constructs.

For this purpose, the abruptions from $Java_I$ are extended to include exceptions. Similar to the abruptions for `return` or `break`, the special value `exc` is used to represent an exception being passed up the source tree.

The addition of generics to the language does not affect $Java_E$, because the Java language does not allow generic exceptions.

2.2.5 Concurrency

$Java_T$ adds concurrency support to the Java language. The submachine $execJava_T$ of *execJava* is used to extend the language with the `synchronized` construct, and to support the `wait()` and `notify()` built-in functions.

Moreover, a new machine *execJavaThread* is introduced which uses the ASM **choose** function to pick one of the running threads (without specifying the actual scheduling algorithm), and then uses the *execJava* machine to run one computation step of that thread.

This model does not handle all cases allowed by the JVM memory model because it does not account for the effects caused by memory local to a processor and instead simply assumes an interleaving of instructions using shared memory. However, for correctly synchronized programs (as defined in the Java memory model), the simplified model agrees with the Java specification.

2.3 Java Generics

Up to Java 1.4, Java did not have any support for generic types. Instead, Java programmers used the type `Object` when writing generic code. However, this made it hard to use generic classes (especially the Java collection library), as any objects read from generic classes had to be manually casted to the appropriate type before they could be used.

In version 1.5, the Java type system was extended with parameterized types. This allows declaring classes or interfaces using type parameters, for example:

```
interface List<T> {
        void add(T element);
        T get(int index);
        ...
}
```

Formally, this is the universal type $\forall T.\texttt{List<}T\texttt{>}$ with type parameter T.

Such an interface can be used by specifying the type parameter, i.e. by choosing the type `Dog` for T:

```
List<Dog> list = new ArrayList<Dog>();
list.add(new Dog());
Dog d = list.get(0);
```

An important consideration in the original design of Java Generics was the ability to retrofit existing class libraries with generic type signatures while keeping compatibility with existing source code and binary class files [LY99] at the same time [BOSW98].

To achieve this compatibility, type information is removed as part of the compilation process. So the code above results in:

```
List list = new ArrayList();
list.add(new Dog());
Dog d = (Dog) list.get(0);
```

The compiler removes all type arguments and uses the *erased* types instead. Moreover, the compiler inserts the appropriate casts when calling methods or reading the value of a `public` field.

The lack of type information at runtime leads to a number of limitations in the language. The most obvious one is the inability to create arrays of generic types. This restriction is necessary because arrays need type information for the runtime checks when storing an element in an array (see Section 3.9 for details).

2.3.1 Wildcards

The types `List<Dog>` and `List<Animal>` are unrelated and there is no conversion from one to the other. Adding such a conversion directly would be unsafe: it is possible to add a `Giraffe` to a `List<Animal>`, so it would be unsafe if a variable of type `List<Animal>` could contain a value of type `List<Dog>` at runtime. For this reason, the original proposal for Java Generics (GJ [BOSW98]) did not include support for such a conversion.

However, it is desirable to allow the use of polymorphism for generic types as well. In 2002, Igaraschi and Viroli suggested an extension to GJ that adds variance-based subtyping [IV02]. This formed the basis for the wildcard feature in Java 5.

Wildcards with upper bounds are a form of use-site covariance. They allow passing generic types to other methods without requiring the type arguments to match:

```
List<? extends Animal> list = new ArrayList<Dog>(...);
Animal a = list.get(0);
```

In this example, the variable `list` can be only used to read from the list. It is not possible to add elements (except `null`) because they would have to be subtypes of the type hidden by the wildcard.

The Java language also allows contravariance by using wildcards with lower bounds. These use the following syntax:

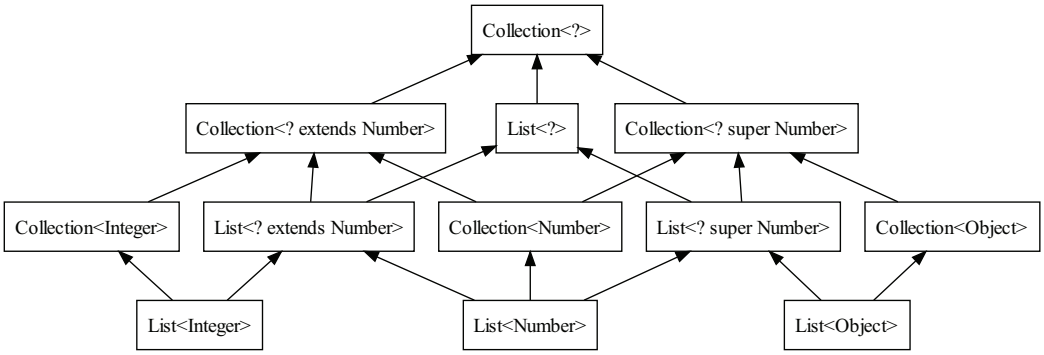

Figure 2.1: Subtyping Graph of Wildcard Types
Arrows point from subtype to supertype.

```
List<? super Animal> list = new ArrayList<Object>(...);
list.add(new Dog());
```

In this case, it is also possible to read from the list, but the return type of the `get` method will be the upper bound of the type parameter of the `List` class, i.e. `Object`.

The Java language disallows wildcards with both upper and lower bounds; but this restriction is unnecessary and does not simplify the language semantics, as type variables with both bounds can occur anyway due to upper bounds declared on the type parameter [TEH05].

Figure 2.1 demonstrates the interaction between subtyping using inheritance ($\texttt{List} \preceq \texttt{Collection}$) and subtyping using wildcard parameterized types.

2.3.2 Wildcard Capture

A unique feature of the Java type system is *wildcard capture*. Wildcard capture allows using wildcards where an actual type is expected, for example when calling a generic method. When capture conversion is used, the wildcard gets replaced by a fresh, global type variable. Unlike wildcards where a different type might hide behind every `?`, the fresh type variables introduced by the compiler allow the type system to tell which unknown types refer to the same type.

The following example from [CDE08] illustrates this language feature:

```
<X> Pair<X, X> make(List<X> x) {}
<X> Boolean compare(Pair<X, X> x) {}
void m(Pair<?, ?> p, List<?> b)
{
```

```
  this.compare(p); //1, type incorrect
  this.compare(this.make(b)); //2, OK
}
```

The first invocation of `compare` is invalid because the two items might have different types: capture conversion converts the type of `p` into `Pair<`Z_1`, `Z_2`>`, where Z_1 and Z_2 denote the fresh type variables.

The second invocation of `compare` is valid because capture conversion occurs when calling `make`, producing the type `List<`Z_3`>`. The type parameter X of the method call is inferred to be Z_3, so return type of the `make` call (and type of the argument being passed to `compare`) is a `Pair<`Z_3`, `Z_3`>`.

There is no direct way for Java programmers to name such a type so that the result of `make` could be stored in a variable before being passed to `compare`. Instead, programmers have to write a helper method to capture the wildcard in a type parameter, giving the type a name that can be denoted in the language.

Parameterized types involving wildcards correspond to existential types. The type `List<? extends Animal>` corresponds to $\exists T \preceq$ `Animal` . `List<`T`>` The *open* operation on existential types corresponds to wildcard capture and is invoked automatically whenever a wildcard type is used. This means `make(b)` roughly corresponds to `open b as [Z,z] in make(z)`. The `open` operation creates a new type variable `Z` and converts the value from `b` (of type `List<?>`) to type `List<Z>`, storing the converted result in the new variable `z` [IV02].

However, there is a major difference between wildcard types and existential types: the fresh type variables introduced by wildcard capture are created in global scope, removing the need for the `close` operation used with existential types.

2.3.3 Generic Methods

Java 1.5 does not only allow parameterized types, but also parameterized methods – these were already used in the examples above.

Generic methods can be declared and called using the following syntax:

```
// declaration
<T extends Comparable<T>> void min(T a, T b) {
  return a.compareTo(b) <= 0 ? a : b;
}
void test() {
  // call with explicit type arguments
  this.<Integer>min(1, 2);
}
```

When calling generic methods, the programmer may leave out the explicit type arguments, and Java will attempt to infer them automatically.

This allows calling the `min` functions as follows:

```
min(1, 2); // T = Integer
min(1.0, 2.0); // T = Double
min(1, 2.0); // T = Number
```

2.3.4 Limitations of Java Generics

In this section, we will describe some of the limitations in the design of Java generics.

Type Erasure

When compiling Java code with generics, the compiler uses *type erasure* to translate the code into equivalent code that does not use generics. This is done to allow adding generics to existing libraries (for example in the collections library) while keeping binary compatibility with old class files referencing those libraries [GJSB05, §4.7].

Type erasure works by replacing all occurrences of parameterized types with the simple class type, and all occurrences of type parameters with their upper bound. Additionally, casts are inserted for the return value of method calls where necessary.

While this was necessary for compatibility, the absence of type information at runtime imposes a few major restrictions on the use of generic types. For example, it is impossible for the Java runtime to check whether an object is of type `List<String>` – the runtime can only verify that the object has the class `List`. As a result of this limitation, the following code compiles successfully (although with an *unchecked warning*):

```
void fillList(Object o) {
    List<Integer> b = (List<Integer>)o; // unchecked warning
    b.add(1);
}
void m() {
    ArrayList<String> a = new ArrayList<String>();
    fillList(a);
    String i = a.get(0); // InvalidCastException
}
```

At the cast in `fillList` an unchecked warning occurs, telling the programmer that the cast will not be checked at runtime. As a result, when executing this code, an integer will be stored in a list of strings. The JLS calls this situation *heap pollution* [GJSB05, §4.12.2.1]. Only later, when reading from the list, this error is detected at runtime by the cast that the compiler inserted after the `get` call.

Unchecked warnings occur whenever one of the constructs that could potentially create a hole in the type system is used. Unfortunately, this also includes array types: Java arrays are covariant and need a runtime type check to throw an `ArrayStoreException` when any reference type is stored in the array. Due to type

erasure, such runtime checks cannot be done for generic types, so creating arrays of parameterized types always causes an unchecked warning.

Wildcard with Lower and Upper Bounds

Here is an example that demonstrates a problem with Java type inference, caused by the lack of wildcards bounded in both directions:

```
interface Animal {}
interface Herbivore extends Animal {}
interface Carnivore extends Animal {}
class Pig implements Herbivore, Carnivore {}
class Test {
    static <T> T pick(T a, T b) { ... }
    public void test(List<Herbivore> h, List<Carnivore> c) {
        Animal a = pick(h, c).get(0); // 1
        pick(h, c).add(new Pig()); // 2
    }
}
```

Using Java 1.5 type inference, the result type of the pick calls is:

```
List<? extends Animal>
```

This makes line 1 (reading `Animal`) valid, but line 2 (writing `Pig`) causes a compile-time error.

It is reasonable to expect that we can add pigs to this list. Pigs are omnivores, so they fit into both lists `h` and `c`. The Java 1.5 type system does not allow expressing a type that can both read animals and write omnivores, but the extension of the type system with wildcards that have both lower and upper bounds allows us to express such a type:

```
List<? extends Animal super Herbivore&Carnivore>
```

2.3.5 Infinite Types

A major problem in the Java type system is the inferred type of the following method call:

```
class Integer implements Comparable<Integer> {}
class String implements Comparable<String> {}

<T> T pick(T a, T b) {}
<T extends Comparable<? extends T>> void use(T a) {}

void test(Integer n, String s) {
```

```
    use(pick(n, s)); // what is the inferred type argument?
}
```

According to the Java Language Specification, the `pick` call returns the least upper bound of Integer and String. Following the definition of *lub* in the JLS [GJSB05, §15.12.2.7], we get:

$$T = lub(\texttt{Integer}, \texttt{String}) = \texttt{Comparable<? extends } lub(\texttt{Integer}, \texttt{String})\texttt{>}$$

This is an infinite type. Thus, the call to `use` is legal: the type returned by pick can indeed be compared to itself.

Note that official Java compiler (version 1.6.0.20), does not implement this correctly. Instead, it infers the following type for the call to `pick`:

$$T = \texttt{Comparable<? extends Comparable<?>>}$$

So the infinite recursion is aborted at the second level, and the compiler uses an unbounded wildcard instead. This makes the call to `use` fail with a type error.

3 Modeling the Type System

This chapter describes the type system used by our implementation of Java Generics for the *execJava* ASM.

There were quite a few possible choices for type systems: [TEH05] and [CDE08] provide formal models for the Java type system, but both ignore some aspects of the language such as type inference or intersection types.

The Java language specification itself is incomplete and does not define some important properties such as the subtyping relationship for infinite types [SC08].

In his Master's thesis [Smi07], Daniel Smith presents two alternatives which improve upon the Java type system by removing some unnecessary restrictions, and allow for an improved type inference algorithm.

3.1 Union-based Type System

The first approach proposed in [Smi07] is to extend the type system to include *union types*. These were first considered as extension to Java by Igarashi and Nagira in [IN06].

In the context of Java generics, union types remove the need for infinite types. Comparing with infinite types in Section 2.3.5, this approach allow representing the result type of the `pick(n, s)` call with the union type `Integer|String`. This is also a valid choice for the type parameter `T` in the call to `use`. Figure 3.1 shows that this union type satisfies the F-bound:

$$\texttt{Integer|String} \preceq \texttt{Comparable<? extends Integer|String>}$$

In the figure, the derivation for $\texttt{S} \preceq \texttt{C<? ext I|S>}$ has been left out; it is analogous to the derivation for $\texttt{I} \preceq \texttt{C<? ext I|S>}$.

However, it is not clear how these union types would interact with other parts of the language. Igarashi and Nagira define the methods of a union type to be the

$$\textsc{Union-Sup}\,\frac{\dfrac{\textsc{Class-Sup}\,\dfrac{}{\texttt{I} \preceq \texttt{C<I>}} \quad \textsc{Class-Contain}\,\dfrac{\textsc{Union-Sub}\,\dfrac{\texttt{I} \preceq \texttt{I}}{\texttt{I} \preceq \texttt{I|S}}}{\texttt{C<I>} \preceq \texttt{C<? ext I|S>}}}{\texttt{I} \preceq \texttt{C<? ext I|S>}} \quad \texttt{S} \preceq \texttt{C<? ext I|S>}}{\texttt{I|S} \preceq \texttt{C<? ext I|S>}}$$

Figure 3.1: Union Type Satisfies F-bound

methods that are common in all types of the union, where methods just need the same name and compatible parameter types. This allows using unions as ad-hoc interfaces, but would complicate the compilation to byte code because runtime type tests would be necessary for each possible type in the union. A nominal approach to define the members of a union type would fit the Java compilation model better.

We found another, related problem when considering the following Java code:

```
interface Animal {}
interface Herbivore extends Animal {}
interface Carnivore extends Animal {}
class Test {
    static <T> T pick(T a, T b) { ... }
    static <T> void print(List<T> list) { ... }
    public void test(List<Herbivore> h, List<Carnivore> c) {
        print(pick(h, c));
    }
}
```

This example is similar to the code in Section 2.3.4. It is valid code accepted by the Java 1.6 compiler, which infers the result of `pick` to be `List<? extends Animal>`. However, in the union-based system, the type `List<Herbivore>|List<Carnivore>`, while being more specific than the Java 1.6 choice, causes a problem for type inference in the `print` invocation.

To allow this case to work, the union type must be converted into a type that does not involve a top-level union, so that the `T` in `print` can be inferred to be one specific type. In this case, the type

```
List<? extends Herbivore|Carnivore super Herbivore&Carnivore>
```

comes to mind as a possible replacement for the union type.

This wildcard type is not equivalent to the original union type: while the original union type guarantees that the lists are homogenouos, the wildcard type also allows instances of the inhomogeneous list `new ArrayList<Herbivore|Carnivore>`.

This approach to replace union types (possibly as part of wildcard capture) would also help defining the members of a union type: the replacement wildcard type is sufficiently specific to fulfill the expectations (read animals, write omnivores).

Due to these open issues and the major differences compared with the type system of the Java language, we decided not to use union types as basis for our ASM implementation.

3.2 Join-based Type System

The second possibility mentioned by Daniel Smith in [Smi07] is to handle infinite types explicitly by using *wildcard references*. Essentially, this approach allows rep-

resenting infinite types by allowing the bound of the wildcard to refer back to any enclosing wildcard.

Wildcard references consist of a single integer, which specifies the index in the list of enclosing wildcards. For example, $?_0$ refers to the innermost wildcard:

```
A<? extends B<? extends ?0>>
  = A<? extends B<? extends B<? extends B<? ...>>>>
```

$?_1$ refers to the wildcard containing the innermost wildcard:

```
A<? extends B<? extends ?1>>
  = A<? extends B<? extends A<? extends B<? ...>>>>
```

The join-based type system stays close to the official Java Specification by making use of infinite types. The type system used in this thesis is closely based on the join-based type system.

3.3 Syntax of Type Declarations

Here we reproduce the syntax for Java class declarations:

ClassDeclaration :
  [*ClassModifiers*] `class` *Identifier* [*TypeParameters*] [*Super*] [*Interfaces*] *ClassBody*

TypeParameters :
  `<` *TypeParameter* { `,` *TypeParameter* } `>`

TypeParameter :
  Identifier [*TypeBound*]

TypeBound :
  `extends` *ClassOrInterfaceType* { `&` *ClassOrInterfaceType* }

Super :
  `extends` *ClassType*

Interfaces :
  `implements` *InterfaceType* { `,` *InterfaceType* }

Figure 3.2: Syntax for Java Class Declarations

The syntax in Figure 3.2 describes class declarations. Each class declares a list of type parameters. Each type parameter in turn declares a set of upper bounds separated by „`&`“. If multiple upper bounds are present, the combined upper bound is represented using an intersection type.

A class also declares its base class and the set of interfaces it implements. Wildcards cannot be used as type arguments passed to the base class – see Section 3.6 for more details.

3.4 Syntax of Type Expressions

This section shows the syntax of type expressions – references to types as used in variable declarations and method signatures.

Type :
  ClassOrInterfaceType {`[]`}
  BasicType {`[]`}

ClassOrInterfaceType :
  Identifier [*TypeArguments*] { `.` *Identifier* [*TypeArguments*] }

TypeArguments :
  `<` *TypeArgument* {`,` *TypeArgument* } `>`

TypeArgument :
  Type
  `?` [(`extends`|`super`) *Type*]

BasicType :
  `byte`
  `short`
  `char`
  `int`
  `long`
  `float`
  `double`
  `boolean`

Figure 3.3: Syntax for Java Type Expressions

The grammar in Figure 3.3 is based on the grammar in the JLS [GJSB05, §18.1] and is simplified a bit by allowing arbitrary types as type arguments. We will use static constraints on types to ensure that only reference types are used as type arguments.

3.5 Types in the Java Type System

The Java type syntax shown above allows primitive types, types named by identifiers, type arguments, and arrays.

However, the Java type system knows additional types that cannot directly be represented in its syntax. For example, type parameters with multiple bounds make

use of intersection types, which are not denotable in Java type expression syntax:

```
<T extends Comparable<T> & Cloneable> T method();
```

To represents all types that occur during calls to generic methods, we need to generalize the definition of types:

1. Primitive types are types.
2. Classes and interfaces are types. If the class/interface is generic, this type is called the „raw type“.
3. `Null` and `void` are types.
4. If A is a type other than `void`, then A[] is a type.
5. If C is generic class with N type parameters, and A_i (i=1..N) are type arguments that satisfy the bounds of the respective type parameters in C, then $C < A_1, ..., A_n >$ is a type. Such a type is called a parameterized type.
6. If all of $A_1, ..., A_n$ $(n > 0)$ are reference types, then $A_1\&...\&A_n$ is a type. Such a type is called an intersection type.
7. Type variables are types. Type variable always refer to a type parameter of an enclosing class or method, and have an upper bound.
8. Captured wildcards are types. These types are introduced by the compiler when calling generic methods or accessing members of generic classes using wildcards as type arguments. Captured wildcards have both an upper and a lower bound.

This definition replaces [SSB01, §5.1.1].

In this definition, „reference types“ is used to refer to all types except for primitive types and `void`.

A *type argument* is one of the following:

1. a reference type
2. a wildcard with both an upper and lower bound: `? extends` U `super` L
3. a wildcard reference: $?_i$

For wildcards without upper bound, U is assumed to be `Object`. For wildcards without lower bound, L is assumed to be the `Null`type.

3.6 Well-Formed Types

Wildcard references can be used in the place of wildcards to represent infinite types, but are well-formed only if there is an enclosing wildcard to be referenced. Thus, `C<?`$_0$`>` is not well-formed, but so is `C<? extends C<?`$_0$`>>`.

For a parameterized type to be well-formed, the number of type arguments must match the number of type parameters declared on the generic class. Moreover, the following conditions must be satisfied for each type argument A_i being applied to the corresponding type parameter X_i:

- If the type argument is a type, it must be within the declared bounds of the type parameter:

$$A_i \preceq \lceil X_i \rceil \, [X_1 := A_1, \ldots X_n := A_n]$$

As every type parameter has an implicit upper bound of `Object`, this implies that type arguments must be reference types.

- If the type argument is a wildcard with upper bound, the bound must not conflict with the declared upper bound of the type parameter:

$$\lceil A_i \rceil \texttt{ \& } \lceil X_i \rceil [X_1 := A_1, \ldots X_n := A_n] \text{ is well-formed}$$

This rule disallows wildcards when no type (except `Null`) exists that matches the wildcard bounds. For example, if both bounds refer to unrelated class types (e. g. $\lceil A_i \rceil =$ `String`, $\lceil X_i \rceil =$ `Number`), no type can fulfill these bounds as Java does not allow multiple inheritance for classes. Also, types involving multiple incompatible instanciations of parameterized types are disallowed (e. g. $\lceil A_i \rceil =$ `Collection<Number>`, $\lceil X_i \rceil =$ `List<Integer>`).

- If the type argument is a wildcard with both lower and upper bounds, the lower bound must be a subtype of the upper bound:

$$\lfloor A_i \rfloor \preceq \lceil A_i \rceil$$

This rule is necessary to prevent types such as

```
C<? extends String super Object>,
```

which would lead to an unsound type system as described in [CDE08].

- If the type argument is a wildcard with lower bound, the lower bound must be a subtype of the declared upper bound of the type parameter:

$$\lfloor A_i \rfloor \preceq \lceil X_i \rceil [X_1 := A_1, \ldots X_n := A_n]$$

For an intersection type A_1 `&` $\ldots$ `&` A_n to be well-formed, it must fulfill these conditions:

- For any two class types A_i and A_j:

$$A_i \preceq A_j \vee A_j \preceq A_i$$

Intersections of two unrelated class types are disallowed as `null` would be the only valid element of this type.

- For any two occurrences of a generic interface in the types of the intersection or their supertypes, the type arguments must be the same. This is because Java does not allow inheriting from multiple different instanciantions of a generic interface.

In the class inheritance declaration, when declaring a base class or interfaces, there are additional restrictions on types. In these cases, only class or interface types are allowed. These can be parameterized, but wildcards cannot be used. This restriction is necessary to ensure that the direct supertype operation ($T\uparrow$, defined in Section 3.10) does not return wildcard parameterized types. Thus, the following type declarations are invalid:

```
abstract class A implements List<?> {}
class B extends A[] {}
class C<T> extends T {}
```

However, within the arguments to parameterized types, all types may be used, so the following declarations are valid:

```
abstract class D implements List<List<?>> {}
class E implements ArrayList<D[]> {}
class C<T> extends ArrayList<T> {}
```

3.7 Parameterized Types

A parameterized type is constructed by passing type arguments to a generic class. The members of the parameterized type are the members of the class, where every occurrence of a type parameter is replaced with the appropriate type argument.

Bounds used in wildcards passed to a parameterized type may refer to the type itself. Such infinite types cannot be written by the programmer using Java syntax, but they can be introduced by type inference. For example, the least upper bound of `Comparable<Integer>` and `Comparable<String>` is `Comparable<? extends ?`$_0$`>` [Smi07].

Only wildcard references can be used to build types that refer to themselves; parameterized types cannot be directly infinite. That is, there is no type such as:

```
Comparable<Comparable<Comparable<...>>>
```

Inside parameterized types using wildcards, these references are represented using an index referring to an enclosing wildcard (see Section 3.2). However, taken on their own, such wildcard references do not constitute well-formed types.

Thus, when accessing the bounds of the wildcard, any wildcard references pointing to this outermost wildcard must be replaced with the wildcard itself.

Given the wildcard

$$W = \texttt{? extends } U \texttt{ super } L,$$

the bounds of the wildcard are defined to be:

$$\lceil W \rceil = unroll(U)|_{[W]}$$
$$\lfloor W \rfloor = unroll(L)|_{[W]}$$

The unroll operation, defined in [Smi07, §5.1.3], replaces any occurrence of a wildcard reference to W within U with the full definition of W.

3.8 Type Erasure

When compiling Java to byte code, parameterized types and type variables get replaced with a simple class type by *type erasure*.

However, type erasure is not only used as part of the compiler implementation, but also in the definition of some type system operations, especially where raw types are involved. It is also necessary to model the runtime restrictions caused by type erasure, for example that type tests caused by casts only check for the erased type.

Here we define $|T|$ to be the erasure of the type T.

- If T is a parameterized type, then $|C\texttt{<}\ldots\texttt{>}| = C$.
- If T is an array type, then $|E\texttt{[]}| = |E|\texttt{[]}$.
- If T is an intersection type, then the erasure is the first type in the intersection: $|T_1\ \texttt{\&}\ T_2| = |T_1|$.
- If T is a type variable or captured wildcard, then the erasure is the upper bound of that variable: $|X| = |\lceil X \rceil|$.
- If T is any other type (class or interface type, or primitive type), then $|T| = T$.

3.9 Reifiable Types

A type is said to be *reifiable* if type erasure does not remove important type information. Some language constructs like casts and stores to arrays require type information at runtime. For example, due to type erasure, there is no way for the Java runtime to verify the store to the array in the following program:

```
Object[] arr = new C<A>[1];
arr[0] = new C<B>();
```

For this reason, the Java allows creating arrays (without unchecked warning) only if the element type is reifiable. Similarly, the `instanceof` operator can only be used with reifiable types, and casts produce an unchecked warning if the target type is not reifiable.

The definition of *reifiable* is given in the JLS [GJSB05, §4.7] by listing the reifiable types. However, we give the definition in terms of type erasure and subtyping, making the requirement for the safety of runtime type checks more clearly visible:

$$T \text{ is reifiable} \iff |T| \preceq T$$

This means any type unchanged by erasure if reifiable; and additionally, parameterized types with unbounded wildcards (`C<?>`) are also reifiable.

3.10 Direct Supertype

Subtyping and some other algorithms like member lookup need to be able to determine the inheritance hierarchy of a parameterized type, including any type parameters. Because type declarations may change the usage of type parameters in every

level of the inheritance hierarchy, determining the direct supertype requires replacing type parameters with arguments on each level.

For example, given the class declaration:

```
abstract class Test<A,B> implements List<Pair<B,A>> {}
```

the direct supertype of `Test<String,Integer>` is `List<Pair<Integer,String>>`.

We will write $T\uparrow$ to denote the direct supertype of type T. The definition is similar to that in [SC08]:

- If $T =$ `Object`, then $T\uparrow$ is undefined.
- If T is a parameterized type C`<`$T_1 \dots T_n$`>`, then the direct supertype is the intersection of the supertypes of `class C<`$X_1 \dots X_n$`> extends` $S_1 \dots S_m$: let σ be the replacement $[X_1 := T_1 \dots X_n := T_n]$; then $T\uparrow = \sigma S_1$ `&` $\dots$ `&` σS_m
- If T is a reference to a class without type parameters, then the direct supertype is the intersection of the supertypes of the declared supertypes: $T\uparrow = S_1$ `&` $\dots$ `&` S_m
- If T is a raw type, then the direct supertype is the intersection of the erasures of the declared supertypes: $T\uparrow = |S_1|$ `&` $\dots$ `&` $|S_m|$

This definition requires that T is either a class or interface type different from `Object`, or that it is a parameterized type which does not have wildcards as arguments. For all other types T, we consider $T\uparrow$ to be undefined.

The supertype of types parameterized with wildcards is left undefined because the definition would involve wildcard capture (the replacement needs to produce a type for the wildcard). Wildcard capture produces a new type variable every time it is invoked, but we do not want two invocations of $T\uparrow$ to produce different results. Instead, we will assume that wildcard capture is invoked before the direct supertype needs to be determined.

Because class declarations are restricted to not use wildcards, direct supertypes cannot be wildcard parameterized types.

3.11 Wildcard Capture

Wildcard capture, called capture conversion in [GJSB05, §5.1.10], corresponds to the *open* operation on existential types. Wildcard capture must be invoked before a member of a parameterized type is accessed because occurrences of the type parameter in method signatures must be replaced with another type, but wildcards on their own are not valid as types.

Wildcard capture transforms a parameterized type with wildcards into a parameterized type without wildcards. For all other types, it acts as an identity conversion.

We will use $\|T\|$ to refer to type resulting from wildcard capture of type T.

Given a parameterized type C`<`$A_1, \dots A_n$`>`, where the class C declares the type parameters $X_1 \dots X_n$, wildcard capture replaces every wildcard A_i by a fresh type variable S_i:

$\|C\texttt{<}A_1, ...A_n\texttt{>}\| = C\texttt{<}S_1, ...S_n\texttt{>}$
The following bounds are used for the new type variables S_i:

- $\lceil S_i \rceil = \lceil A_i \rceil$ `&` $\lceil X_i \rceil [X_1 := S_1, \dots X_n := S_n]$
- $\lfloor S_i \rfloor = \lfloor A_i \rfloor$

This means the upper bounds of the new type variable is the intersection type of the wildcard's upper bound and the type parameter's upper bound. The type `Object` is used instead of $\lceil A_i \rceil$ or $\lceil X_i \rceil$ if the wildcard or the type parameter does not declare an upper bound.

The replacement $[X_1 := S_1, \dots X_n := S_n]$ is used to deal with F-bounded type parameters: when a type parameter is bounded by itself, the newly introduced type variable cannot simply copy that bound, but needs to replace the type parameter with the fresh type variable that we just created.

3.12 The Subtype Relation

The subtype relation $\preceq$ is defined using the following rules:

$$T \preceq T \quad \text{(Reflex)}$$

$$\frac{S \preceq S' \quad S' \preceq T}{S \preceq T} \quad \text{(Trans)}$$

$$null \preceq T \quad \text{(Null)}$$

$$\frac{\forall i \leq n, S_i \cong T_i}{C\texttt{<}S_1 \dots S_n\texttt{>} \preceq C\texttt{<}T_1 \dots T_n\texttt{>}} \quad \text{(Class-Equiv)}$$

$$\frac{\forall i \leq n, S_i \in W_i}{C\texttt{<}S_1 \dots S_n\texttt{>} \preceq C\texttt{<}W_1 \dots W_n\texttt{>}} \quad \text{(Class-Contain)}$$

$$\frac{\|C\texttt{<}W_1 \dots W_n\texttt{>}\| \preceq T}{C\texttt{<}W_1 \dots W_n\texttt{>} \preceq T} \quad \text{(Class-Capt)}$$

$$C\texttt{<}T_1 \dots T_n\texttt{>} \preceq C \quad \text{(Class-Erase)}$$

$$C \preceq C\texttt{<?,...?>} \quad \text{(Class-Erase-Unbounded)}$$

$$T \preceq \|T\| \uparrow \quad \text{(Class-Sup)}$$

$$T\texttt{[]} \preceq \texttt{Cloneable \& Serializable} \quad \text{(Arr-Class)}$$

$$\frac{S \preceq T}{S\,\texttt{[]} \preceq T\,\texttt{[]}} \quad \text{(Arr-Covar)}$$

$$X \preceq \lceil X \rceil \quad \text{(Var-Sup)}$$

$$\lfloor X \rfloor \preceq X \quad \text{(Var-Sub)}$$

$$T_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T_n \preceq T_i \quad \text{(Inter-Sup)}$$

$$\frac{\forall i \leq n, S \preceq T_i}{S \preceq T_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T_n} \quad \text{(Inter-Sub)}$$

These rules correspond to the ones found in [SC08], except that the rules related to union types were removed and the rule Class-Erase-Unbounded was added. The latter allows implicit conversions from a raw type to a parameterized type with unbounded wildcards and is important for the definition of reifiable types (Section 3.9).

In the subtyping rules, $\cong$ is used to refer to type equivalence:

$$S \cong T \iff S \preceq T \wedge T \preceq S$$

The symbol $\in$ is used to mean that a type is contained within the bounds of a wildcard:

$$T \in W \iff \lfloor W \rfloor \preceq T \wedge T \preceq \lceil W \rceil$$

An algorithmic implementation of subtyping based on these rules is problematic in the presence of infinite types. In fact, Kennedy and Pierce have shown that a generalized version of the problem is undecidable [KP06]. It is currently still unclear whether subtyping in the Java language with its restriction on multiple implementation inheritance is decidable.

We refer to [Smi07, §5.1.4] for a description of the subtyping algorithm.

3.13 Join Function

The join function calculates the most specific common supertype of two types. It is used in the definition of the conditional operator (`a?b:c`) and in type inference for generic method calls.

The JLS calls this the *least upper bound* (*lub*) [GJSB05, §15.12.2.7]. It is defined by finding the common erasures of all supertypes of the two input types, and then reconstructing the type arguments for them. To reconstruct the type arguments, the algorithm creates wildcard parameterized types whenever two input types provide different type arguments to the same class. To determine the upper bound of this wildcard, the join function is invoked recursively on the type arguments. This recursion can be infinite, producing an infinite type.

The definition of the join function in the JLS has major limitations: due to the definition based on erasure, it can work only with class or interface types, and will produce too imprecise bounds when invoked on type variables [SC08].

```
<T> T pick(T a, T b) { ... }

<C, A extends C, B extends C>
C test(A a, B b) {
  return pick(a, b); // inferred type: Object
}
```

Here, Java infers the type `Object` even though `C` would have been a better choice. Moreover, in some cases it is not clear whether *join* should produce a wildcard with lower or upper bounds - in this case, the extension of the type system to wildcards with both bounds helps avoiding the ambiguity (see Section 2.3.4).

The inability to represent wildcards with both bounds also causes another problem: Type inference expects that the *lub* function is commutative and associative. However, the information loss due to discarding the lower bound causes the *lub* function to lose the associativity.

$$
\begin{aligned}
&lub(lub(\texttt{C<? super T>}, \texttt{C<T>}), \texttt{C<S>}))\\
&\quad = lub(\texttt{C<? super T>}, \texttt{C<S>})\\
&\quad = \texttt{C<? super T\&S>}
\end{aligned}
$$

$$
\begin{aligned}
&lub(\texttt{C<? super T>}, lub(\texttt{C<T>}, \texttt{C<S>}))\\
&\quad = lub(\texttt{C<? super T>}, \texttt{C<? extends Object>})\\
&\quad = \texttt{C<?>}
\end{aligned}
$$

Because the function is applied repeatedly on an unordered set (in the *lci* helper function), this causes type inference to non-deterministically choose one of the options.

As part of the join-based type system, Smith defines the *join* function based on the subtyping relation, allowing it to handle all cases including type variables and array types [Smi07]. Because it uses wildcards with both upper and lower bounds, the *join* function is not affected by the shortcomings of the JLS definition.

Similar to the JLS algorithm, it is possible that infinite types are created (compare with Section 2.3.5). In this case, the algorithm detects when a join invocation matches another invocation that is active, and produces the appropriate wildcard reference to refer to the result of the outer invocation of the join function.

4 ASM Model for Generic Java

The Jbook uses the term *elaboration phase* to refer to transformations done on the syntax tree before the Java program can be executed.

The elaboration phase corresponds to algorithms executed at compile-time in an actual Java implementation. The most significant part of the elaboration phase is the processing of method invocations to determine the target method (overload resolution) and the type of the call (static, virtual, etc.).

The Jbook uses a simple pseudo-code algorithm to describe the compile-time handling of methods [SSB01, §4.1.6]. The introduction of generics and type inference significantly increases the complexity of these operations. This makes it desirable to use the expressive power of Abstract State Machines not only for the dynamic semantics, but also to model the compile-time overload resolution.

4.1 Assigning Types to Expressions

The Jbook uses the static function $\mathcal{T}$ to assign a type to each expression. In the book, $\mathcal{T}$ is defined in the form of tables that specify the static type for the various expressions.

With generics and type inference, determining the type of an expression is significantly more complicated. Determining types is necessary during the elaboration phase: the chosen overload for method calls depends on the type of the target expression and argument expressions. Likewise, the type of a field depends on the type of the reference to the class, as type parameters in the declared type of the field need to be replaced. Of course, to choose an overload or determine the target class of an instance field access, it is necessary to infer the type of sub-expressions (the target object reference, and the method arguments). This means that overload resolution and type inference cannot be performed independently, but need to be performed in a single pass over the syntax tree.

4.2 The Elaboration Machine

To model type inference in the ASM, we change $\mathcal{T}$ to be a dynamic function, which will be calculated by the *elabJava* machine. This new machine is not part of the machines representing the dynamic semantics, but instead is used to model the static semantics. Similar to *execJava* in [SSB01, §3.2], we use the *pos* variable to point to the node of syntax tree that needs to be processed next, or to the node that just was processed.

Because $\mathcal{T}$ is initially **undef** everywhere, we can check its value to test whether a node was already processed. If the current node was already assigned a type, we let the machine go one level up in the syntax tree. We also apply wildcard capture to the type of the expression to implement the implicit capture conversion that Java applies to every expression.

if $pos \neq firstpos \wedge \mathcal{T}(pos) \neq$ **undef then**
  $\mathcal{T}(pos) := \|\mathcal{T}(pos)\|$
  $pos := up(body/pos)$

When an expression consists of sub-expressions, those sub-expressions are given types first:

case $body/pos$ **of**
  $^{\alpha}exp_0\ bop\ ^{\beta}exp_1 \rightarrow$
    if $\mathcal{T}(\alpha) =$ **undef then** $pos := \alpha$
    elseif $\mathcal{T}(\beta) =$ **undef then** $pos := \beta$
    else $\mathcal{T}(pos) := table_{3.3}(bop, \mathcal{T}(\alpha), \mathcal{T}(\beta))$

Here, $table_{3.3}$ refers to table 3.3 (binary operators on primitive types) in [SSB01].

In the following, we will assume that sub-expressions are implicitly given types first, allowing us to only write the interesting last case which actually assigns the type.

Figure 4.1 shows the machine $elabJava_I$, which assigns types to $Java_I$ programs.

This corresponds to table 3.4 in the Jbook, except that the rules of the conditional operator were changed in Java 1.5 to support auto-boxing and determining the most specific common supertype of generics. For the list of special cases related to auto-boxing, see [GJSB05, §15.25]. In general, the conditional operator uses the join function to calculate the most specific common supertype of both expressions.

For $elabJava_C$, the elaboration machine gets more complex: to access a static field, the field must be searched in the target class and its base classes. Because Java disallows using type parameters in the type of static fields, we do not need to deal with generics at this point. On the other hand, when calling methods, we cannot

$elabJava_I =$ **case** $body/pos$ **of**
  lit $\rightarrow \mathcal{T}(pos) := \mathcal{T}(lit)$
  loc $\rightarrow \mathcal{T}(pos) := declaredType(loc)$
  $uop\ ^{\alpha}e$ $\rightarrow \mathcal{T}(pos) := table_{3.2}(uop, \mathcal{T}(\alpha))$
  $^{\alpha}e_0\ bop\ ^{\beta}e_1$ $\rightarrow \mathcal{T}(pos) := table_{3.3}(bop, \mathcal{T}(\alpha), \mathcal{T}(\beta))$
  $^{\alpha}e_0 = {}^{\beta}e_1$ $\rightarrow \mathcal{T}(pos) := \mathcal{T}(\alpha)$
  $^{\alpha}e_0\ ?\ ^{\beta}e_1 : {}^{\gamma}e_2$ $\rightarrow \mathcal{T}(pos) := \mathcal{T}(JLS_{15.25}(\mathcal{T}(\beta), \mathcal{T}(\gamma)))$

Figure 4.1: Elaboration ASM for Imperative Java

$elabJava_C$ = **case** *body*/*pos* **of**
  $C.Field \rightarrow$
    $\mathcal{T}(pos) := declaredType(findField(C, Field))$
  $C.Method(exps) \rightarrow$
    $letmsig = findOverloadWithTypeInference(C, Method, \mathcal{T}(exps), expectedType(pos))$
    $letTA = inferredTypeArguments(msig)$
    $body := body[C. < TA > msig(exps)/pos]$
    $\mathcal{T}(pos) := declaredReturnType(method)[TP/TA]$
  $C. < TA > Method(exps) \rightarrow$
    $letmsig = findOverload(C, Method, TA, \mathcal{T}(exps))$
    $body := body[C. < TA > msig(exps)/pos]$
    $\mathcal{T}(pos) := declaredReturnType(method)[TP/TA]$

Figure 4.2: Elaboration ASM for Procedural Java

simply use the declared return type of the method, but need to replace the type parameters with the type arguments. At the same time, we modify the body and replace the method calls with another version that includes the type arguments and full method signature (msig). Figure 4.2 shows the definition of $elabJava_C$.

4.3 Method Invocation

Method invocation expressions can have any of the following forms:

1. `Identifier ( ArgumentList )`
2. `Expression . [TypeArguments] Identifier ( ArgumentList )`
3. `TypeName . [TypeArguments] Identifier ( ArgumentList )`
4. `super . [TypeArguments] Identifier ( ArgumentList )`

To determine the method being called, both overloading and overriding needs to be resolved. The former is being resolved at compile-time, while the latter is resolved at run-time because it depends on the run-time type of the target object.

As a first step when resolving overloading, the type to search is determined:

- In the first form (`Identifier ( ArgumentList )`), the target type is the current class (the class containing the method call).
- In the second form, the target type is $\mathcal{T}(Expression)$.
- In the third form, the target type is the class named by the *TypeName*.
- In the last form, the target type is the base class of the current class.

Then, a set of candidate methods is formed by searching for all accessible methods with the name *Identifier* within the target type. Overload resolution is used to identify a single method out of this candidate set.

4.4 Candidate Methods

The set of candidate methods is determined by searching for all methods on the target type, and filtering the resulting list for those which have the requested name and are accessible. For accessibility, the definition [SSB01, §4.1.5] can be used without changes.

When determining the methods on a generic type, all occurrences of the type parameter in the method signature need to be replaced with the type arguments. In order for this replacement to be possible, all type arguments must be types, not wildcards. Because the target type of method resolution is either a raw type or the type of a sub-expression, and because $T\uparrow$ never produces wildcard types, it is safe to assume that wildcard capture has already occurred:

$$\begin{aligned} &\mathit{allMethods}(C\texttt{<}T_1, \ldots, T_n\texttt{>}) = \\ &\quad \mathit{methods}(C)[X_1 := T_1, \ldots, X_n := T_n] \cup \mathit{allMethods}(C\texttt{<}T_1, \ldots, T_n\texttt{>}\uparrow) \end{aligned}$$

In the case of raw types, all type parameters are replaced with the type `Object`:

$$\begin{aligned} &\mathit{allMethods}(C) = \\ &\quad \mathit{methods}(C)[X_1 := \texttt{Object}, \ldots, X_n := \texttt{Object}] \cup \mathit{allMethods}(C\uparrow) \end{aligned}$$

Note that overridden methods are considered only once in the candidate set.

4.5 Overload Resolution

The input to the overload resolution is a set of candidate methods and the method call that needs to be resolved. More precisely, the method call can be represented by the types of the arguments expressions, the type arguments provided by the programmer, if any, the expected type (only in some contexts, see section on type inference for details).

The output is a single method among the candidate methods, or a compile-time error.

Overload resolution in the Java language works in three steps:

1. Type Inference
2. Applicability Testing
3. Choosing the most specific method

In the first step, types are inferred for calls to generic methods if the programmer did not provide explicit type arguments. This is done separately for each candidate method. Type inference never fails, but may infer incorrect types. In the second step, the method is tested for *applicability*. This removes all inapplicable methods from the candidate set and keeps only those where the given arguments and given/inferred type arguments are valid for the method definition. In the third step, the most specific method is chosen from the applicable methods. If no such method exists, a compile-time error occurs.

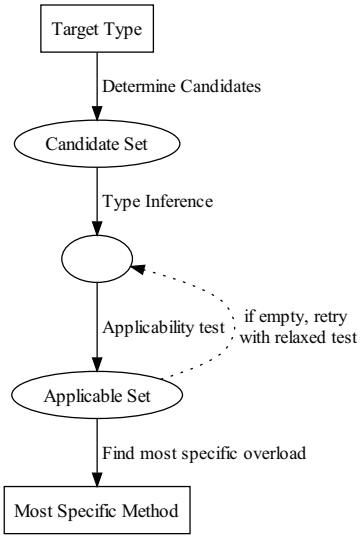

Figure 4.3: Overload Resolution
Boxes indicate single items, circles indicate sets.

The algorithm is further complicated by the addition of auto-boxing and variable arity methods in Java 5. To keep backward compatibility for existing Java source code, the addition of the new features must not cause the compiler to use a different overload or report a call as being ambiguous when the previous Java versions accepted the same source code. To ensure this, the applicability test is executed up to three times: first, it uses a backward-compatible version which does not support auto-boxing or variable arity methods. Only if that test ends up with no applicable methods, the relaxed conversion rules of auto-boxing are used instead. If that second test still ends up with no applicable methods, the applicability rules are relaxed further to allow variable arity methods.

Figure 4.3 shows an overview of the algorithm.

4.5.1 Type Inference

The input to the JLS type inference algorithm is described by a set of constraints that the inferred type should fulfill. For every argument with type A_i passed to a formal parameter of type F_i, a constraint of the form $A_i <:_? F_i$ is created.

Moreover, if the context of the method call expects a specific type, a constraint of the form $E :>_? R$ is created, here E is the expected type and R is the return type of the method. The expected type is only used in cases where it is explicitly specified by the programmer and thus can be determined independently of the overload chosen for the method call. For example, the expected type is inferred in an return or assignment statement.

The notation $A <:_? F$ is used to describe the constraint that, after the type parameters in F are replaced with the inferred types, the type A should be a subtype of F. Based on the subtyping relation, the constaint formula is decomposed into other constraint formulas until the right-hand side is a single type parameter. For example, `List<String>` $<:_?$ `List<T>` results in `String` $<:_? T \wedge$ `String` $:>_? T$. The definition of $<:_?$ and $:>_?$ in [Smi07] also allows for disjunctions, which are used when there are several potentially valid choices for a type parameter. In that case, the whole constraint formula is converted into disjunctive normal form. For each conjunction in the constraint formula, type inference tries to fulfill these bounds by choosing the type arguments to be the lower bounds. If this fails, inference proceeds with the next conjunction.

A full definition of the algorithm is given in [Smi07].

4.5.2 Applicability Testing

Applicability testing is the process of checking whether the specified arguments are valid for an overload. There are multiple versions of the algorithm: support for autoboxing can be enabled or disabled; and variable arity method support is optional in the same way. Java makes use of all four possible combinations – three combinations are used for the three attempts to find an applicable set; and the last is used in the definition of the most specific method.

For an overload to be applicable, all of the following conditions must hold:

- If the method is generic, the number of type parameters must match the number of type arguments.
- If method has variable arity and variable arity is enabled, then the number of arguments must be at least $n-1$, where n is the number of formal parameters. Otherwise, if the method does not have variable arity or if variable arity is disabled, then the number of arguments must be equal to the number of formal parameters.
- If the method is generic, the type arguments must fulfill the bounds of the type parameter:

  $$T_i \preceq \lceil X_i \rceil \, [X_1 := T_1, \ldots, X_p := T_p]$$

  Here, $X_1, \ldots, X_p$ denotes the type parameters and $T_1, \ldots, T_p$ refers to the type arguments. To handle F-bounded type parameters, we substitute the type arguments for the type parameters.

We will call the types of the formal parameter types F_i ($1 \leq i \leq n$, after substituting X_i with T_i) and the types of the argument A_i ($1 \leq i \leq m$).

If method has variable arity and variable arity is enabled, then, in addition to the conditions above, the following must hold:

- $\forall_{1 \leq i \leq n-1} A_i \triangleleft F_i$
- If $n = m$, then $A_n \triangleleft F_n \vee A_n \triangleleft \mathit{element}(F_n)$
- If $n \neq m$, then $\forall_{n \leq i \leq m} A_i \triangleleft \mathit{element}(F_n)$

Here, $element(F_n)$ refers to the element type of the last formal parameter, which always is an array type for variable arity methods. If the method does not have variable arity or variable arity is disabled, then we simply require

$$\forall_{1 \le i \le n-1} A_i \lhd F_i$$

If auto-boxing is disabled, $\lhd$ is the subtype relation $\preceq$. If auto-boxing enabled, then $\lhd$ includes boxing/unboxing conversions:
$A \lhd B =$ **if** B is reference type **then** $box(A) \preceq B$
else $unbox(A) \preceq B$

The function *box* and *unbox* return the wrapper class for the corresponding primitive type and vice versa. All other types are returned unchanged.

Note that having an applicable overload is a necessary but not sufficient condition for a method invocation to be valid. There are additional constraints which are checked after overload resolution.

4.5.3 Most Specific Method

A method is the *most specific* method if it is *more specific* than all other applicable methods.

A method is said to be *more specific* than another method if every call to the first method is also a valid call to the second method. Even though choosing the most specific method occurs as part of overload resolution, the definition of 'more specific' is independent of the call being looked up: being 'more specific' is a relation among method *definitions*.

M_A is more specific than M_B iff M_B is (after type inference) applicable for the call `M(`F_{A1}`, ..., `F_{An}`)`, where F_{Ai} are the types of the formal parameters of M_A.

The applicability test for this purpose allows variable arity methods, but not auto-boxing. This means that even though a call `m(1);` prefers `void m(int a)` over `void m(Object a)` due to the backwards compatibility rule, neither method definition is more specific than the other. This leads to the surprising result that adding the unrelated variable-arity parameter `String... b` to both overloads causes the call to become ambiguous.

This definition for the most specific methods makes type inference necessary even if the input program uses explicit type arguments for every method call.

4.5.4 Static Constraints on Method Calls

When overload resolution succeeds and a single applicable method is chosen, a number of additional constraints must be fulfilled in order for the method call to be valid. The constraints depend on the form of the invocation expression:

- `Identifier ( ArgumentList )`
  In this form, it is an error if the target method is an instance method but the call appears in a static context (static method or field initializer).

- `Expression . [TypeArguments] Identifier ( ArgumentList )`
  In this form, the target method must be an instance method.
- `TypeName . [TypeArguments] Identifier ( ArgumentList )`
  In this form, the target method must be a static method.
- `super . [TypeArguments] Identifier ( ArgumentList )`
  In this form, the target method must be a non-abstract instance method, and the call must not appear in a static context.

4.6 Dynamic Semantics

Storing generic type information in the model for *execJava* is not strictly necessary. Java compilers use type erasure, so generic type information is not available at runtime on most Java implementations. However, for the purpose of modeling the Java semantics, we store the complete type (with type arguments) in each object instance and generic method calls. This allows using the model to show the type safety of Java, and that the casts inserted by the compilation schema will always succeed at runtime (provided that no unchecked constructs were used).

4.6.1 Instances of Generic Classes

To model objects with generic types in the ASM, we change the definition of the heap [SSB01, §5.1.8] to store a type instead of the class.

$$\textbf{data } \mathit{Heap} = \mathit{Object}(\mathit{Class}, [\mathit{Type}], \mathit{Map}(\mathit{Class}/\mathit{Field}, \mathit{Val}))$$

The list of types represents the type arguments that are passed to the class. In addition to the *classOf* function defined in the Jbook (which simply returns the class of an object), we define a *typeOfRef* function to return the complete type:

$$\begin{aligned}&\mathit{typeOfRef} : \mathit{Ref} \to \mathit{Type}\\ &\mathit{typeOfRef}(\mathit{ref}) = \textbf{case } \mathit{heap}(\mathit{ref}) \textbf{ of}\\ &\quad \mathit{Object}(C, \overline{X}, \mathit{fields}) \to C\texttt{<}\overline{X}\texttt{>}\end{aligned}$$

When a new object instance is being created, the type arguments are being specified as part of the source tree. The Java syntax disallows wildcards in this context, so to retrieve the actual type from the type declared in the source code, we only need to replace any type parameters by their current type arguments. For this purpose, both type parameters in the current class and in the current method need to be considered:

$$\begin{aligned}\mathit{newObjectTypeArguments} = [\ &\mathit{lookupTypeInClass}(\mathit{lookupTypeInMethod}(T))\\ &\mid T \in \mathit{declaredTypeArguments}]\end{aligned}$$

lookupTypeInMethod is defined in Section 4.6.2.

$$
\begin{array}{l}
\mathit{lookupTypeInClass}(T) = \\
\quad \textbf{if } \mathit{isStatic}(\mathit{meth}) \textbf{ then } T \\
\quad \textbf{else let } A = \mathit{typeArgumentsForClass}(\mathit{typeOfRef}(\mathit{locals}(\mathit{this})), \mathit{currentClass}), \\
\qquad\qquad X = \mathit{typeParameters}(\mathit{currentClass}) \\
\qquad T[X_1 := A_1, \ldots, X_n := A_n]
\end{array}
$$

This replaces all occurrences of the type parameters of the current class with the type arguments that were used to create the current object instance.

The function *typeArgumentsForClass* here determines the type parameters as they are passed to the current class, if the type of the `this` reference is a subclass of the current class.

$$
\begin{array}{l}
\mathit{typeArgumentsForClass}(T, C) = \textbf{if } |T| = C \textbf{ then } \mathit{typeArguments}(T) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{typeArgumentsForClass}(T{\uparrow}, C)
\end{array}
$$

Note that this replacement of type parameters with arguments is not sufficient to ensure that object instances only use class or interface types: it is still possible for type variables to occur as actual type arguments. For example, it is possible to create an instance of `ArrayList<`Z`>` where Z is a captured wildcard:

```
void main() {
    ArrayList<?> list = createList();
}
<T> ArrayList<T> createList() {
        return new ArrayList<T>();
}
```

4.6.2 Generic Method Calls

To represent the type arguments passed to generic methods at runtime, the stack frames used by the $\mathit{execJava}_C$ machine need to be adjusted. Similar to the existing variables ($\mathit{meth}, \mathit{restBody}, \mathit{nextPos}, \mathit{locals}$), we define a new global variable *methTypeArgs* representing the type arguments passed to the current method. In the Java_C *invokeMethod* rule, the old type arguments are saved to the stack frame, and *methTypeArgs* is set to the type arguments passed to the new method being invoked. In the *exitMethod* rule, the type arguments are restored from the stack frame.

The type arguments being passed to the method being called are those specified by the programmer or those produced by type inference as part of the elaboration step. In both cases, type parameters are replaced with the actual values passed to those parameters.

$$
\begin{array}{l}
\mathit{callTypeArguments} = [\ \mathit{lookupTypeInClass}(\mathit{lookupTypeInMethod}(T)) \\
\qquad\qquad\qquad\qquad | T \in \mathit{declaredTypeArguments}]
\end{array}
$$

The definition of *lookupTypeInMethod* is straight-forward: all occurrences of the type parameters of the current method are replaced with the values stored in the *methTypeArgs* variable.

$$\begin{aligned}&\mathit{lookupTypeInMethod}(T) =\\ &\quad \textbf{let } X = \mathit{typeParameters}(\mathit{meth})\\ &\quad T[X_1 := \mathit{methTypeArgs}_1, \ldots, X_n := \mathit{methTypeArgs}_n]\end{aligned}$$

Together, these changes represent a form of reified generics, which could be used to fix some of the problems associated with type erasure. However, the implementation used in our ASM model is constructing types at runtime (when replacing type parameters with their arguments), creating a performance penalty for generics. This makes this implementation unattractive for use in an actual Java virtual machine.

4.6.3 Simulating Type Erasure

The ASM model simulates the effects of type erasure by ignoring the reified type arguments and instead using the erasure of the compile-time types for runtime operations such as casts and array creations.

For example, the ASM rule for the cast instruction is almost unchanged compared to the original in the Jbook, comparing the class (not the type) of the reference with the erasure of the type specified by the programmer:

$$(T)^{\blacktriangleright}\mathit{ref} \rightarrow \textbf{if } \mathit{ref} = \mathit{null} \vee \mathit{classOf}(\mathit{ref}) \preceq |T| \textbf{ then } \mathit{yieldUp}(\mathit{ref})$$

Similarly, the result of the compilation process is simulated by inserting such a cast at the end of each method invocation. This ensures our model produces InvalidCastExceptions at the same place as the actual Java code. This is achieved by modifying the definition of *exitMethod* (in Figure 4.4) to perform a test whether the returned value matches the expected static type at the call site.

4.7 Type Safety

Obviously, Java in its current form with raw types and unchecked warnings is not type-safe. But even in the absense of unchecked warnings, specification bugs and compiler bugs can cause issues with type-safety.

The Java specification declares that type inference is not required to produce exact results, as these are verified later as part of the applicability check. However, a part of the type inference algorithm, the join function (least upper bound in JLS), is also used in the definition of the conditional operator without any such safety checks.

For example, the JLS definition of *join* incorrectly defines:

$$\mathit{join}(\texttt{List<? extends Number>}, \texttt{List<? super Number>}) = \texttt{List<Number>}$$

Using the conditional operator, this can be used to construct a hole in the type system:

$exitMethod(result) =$
  let $(oldMeth, oldPgm, oldPos, oldLocals, oldMethTypeArgs) = top(frames)$
  $meth := oldMeth$
  $pos := oldPos$
  $locals := oldLocals$
  $methTypeArgs := oldMethTypeArgs$
  $frames := pop(frames)$
  if $methNm(meth) =$ `<clinit>` $\wedge\ result = Norm$ **then**
    $\vdots$
  elseif $(result \in \{Norm, null\} \cup \text{abruptions}) \vee classOf(result) \preceq |\mathcal{T}(oldPos)|$ **then**
    $restbody := oldPgm[result/oldPos]$;
  else
    $restbody := oldPgm[$`throw new InvalidCastException();`$/oldPos]$;

Figure 4.4: ASM Rule for Leaving a Method Invocation

```
List<Number> hole(List<? super Number> input) {
  List<? extends Number> tmp = null;
  return input != null ? input : tmp;
}
void test() {
  List<Object> list = new ArrayList<Object>();
  list.add("Text");
  List<Number> list2 = hole(list);
  // now a List<Number> contains a string
}
```

Fortunately, the Java compiler does not implement this specification bug.

Ignoring such specifications bugs, generic Java (without unchecked warnings) appears to be type-safe. In [CDE08], type soundness is demonstrated for a subset of Java which includes wildcards. However, for a full proof of type safety, it is necessary to solve the problem of reification first in order to make the type system aware of generic runtime types.

The model used in this thesis is a first step in this direction, as it assigns generic types to runtime objects. However, the model as presented here does not handle wildcard capture correctly.

The following example from the EGO reification model [CV08] demonstrates this deficiency:

```
static <T> List<T> clone(List<T> l) {
  List<T> newList = new List<T>();
  newList.head = l.head;
  newList.tail = l.tail;
```

```
  return newList;
}
...
public static void main(String[] args) {
  List<?> l = new List<String>();
  List<?> l2 = clone(l);
}
```

The runtime-type of `l` is `List<String>`. When cloning this list, a programmer would expect that the copy is a `List<String>` as well, but our model produces `List<Z>` where Z is a fresh type variable introduced by wildcard capture. The problem for type safety already occurs within the `clone` method: because `T` is inferred to be `Z`, the parameter `l` of type `List<Z>` contains a value of type `List<String>`, but `ListNode<String>` is not a subtype of `ListNode<Z>`. The definition of type safety as used in the Jbook [SSB01, §8.4], namely that the type of the value computed by any expression is a subtype of the static type of that expression, would not hold with this model.

A possible solution to this problem is to avoid using type variables at runtime and instead pass the type hidden by the wildcard – this is the approach taken by the EGO reification model [CV08]. In the case where there is no type hidden by the wildcard, for example because the runtime value is `null`, it is possible to use the lower bound of the wildcard instead. The lower bound can be safely used without having to deal with F-bounded types parameters because type parameters can only have upper bounds.

5 Implementation with AsmGofer

AsmGofer [Sch01] is a programming system that adds support for Abstract State Machines to the functional programming language Gofer. Gofer is a subset of the popular Haskell programming language.

The CD accompanying the Jbook contains an implementation of the *execJava* ASM in AsmGofer. Static contraints, the elaboration phase and derived functions used by the ASM are implemented as Gofer functions.

For experimenting with the Java model, the Jbook AsmGofer code contains a graphical user interface that allows to single-step through the execution of *execJava*, allowing inspection of the model by stepping through the execution and seeing how expressions are replaced with values.

Figure 5.1 shows the GUI executing a simple test program for `ArrayList`. We have extended the bottom left view to show the reified runtime type of the heap objects, based on the model from Section 4.6.1.

The user interface also allows selecting any expression and pressing „Show type“ to display the static type of that expression. For method calls, this will also display the chosen overload. This allows easily exploring the results of the elaboration phase.

5.1 Parsing Java 1.5

The parser originally used with the Jbook ASM implementation was created using the Happy Parser Generator [Mar]. However, due to the large syntactic changes in Java 1.5, we decided to not extend this parser, but replace it with a different existing parser which can handle Java 1.5.

For this, we chose the open-source project JavaParser [Ges], which is implemented in Java and produces a set of Java objects representing the abstract syntax tree. To convert this AST into the Gofer data structure, we wrote a small program that traverses the AST and outputs Gofer code. When executed, the generated Gofer code reconstructs the syntax tree. This allows using the new parser with the existing abstract source tree defined by the Gofer code included with the Jbook. We only had to perform minor adjustments to the data structures to add support for the new language constructs.

5.2 Implementation of the Type System

To implement the generic type system, we extended the definition of `JavaType` to include all type expressions necessary to represent Java types, including those which

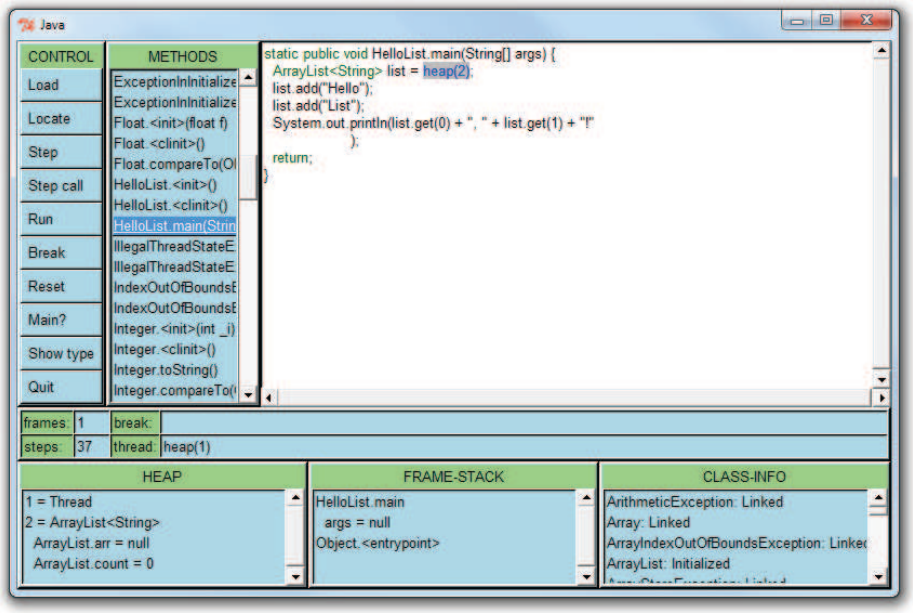

Figure 5.1: AsmGofer GUI for Java ASM

cannot be named directly by the programmer.

$$
\begin{aligned}
\textbf{data}\ \mathit{JavaType} = {} & \mathit{TJInt} \\
& \vdots \\
\mid {} & \mathit{TJBoolean} \\
\mid {} & \mathit{TJRef}(\mathit{TypeName}) \\
\mid {} & \mathit{TJP}(\mathit{TypeName}, [\mathit{TypeArgument}]) \\
\mid {} & \mathit{TJArray}(\mathit{JavaType}) \\
\mid {} & \mathit{TJVariable}(\mathit{Int}, \mathit{TypeParameterOwner}, \mathit{TypeContext}) \\
\mid {} & \mathit{TJFBoundRef}(\mathit{Int}) \\
\mid {} & \mathit{TJCapturedWildcard}(\mathit{Int}, \mathit{JavaType}, \mathit{JavaType}) \\
\mid {} & \mathit{TJIntersection}([\mathit{JavaType}]) \\
\mid {} & \mathit{TJNull} \\
\mid {} & \mathit{TJVoid} \\
\mid {} & \mathit{TJNoType}
\end{aligned}
$$

This declaration defines all the possible types. These correspond to the types defined in Section 3.5.

The constructor *TJRef*, which was already used in the existing implementation, refers to a class or interface type without any type arguments.

We added the constructor *TJP* to refer to a parameterized type. Each type argument is either another JavaType, a wildcard bounded in both directions, or a wildcard reference $?_i$.

$$
\begin{aligned}
\textbf{data}\ \mathit{TypeArgument} = {} & \mathit{TAType}(\mathit{JavaType}) \\
\mid {} & \mathit{TAWildcard}(\mathit{JavaType}, \mathit{JavaType}) \\
\mid {} & \mathit{TAWildcardRef}(\mathit{Int})
\end{aligned}
$$

Wildcard references can occur only within the upper bound of an enclosing *TAWildcard*. When accessing such an upper bound using the $\lceil\ \rceil$ function, wildcard references are replaced as described in Section 3.7, thus ensuring that wildcard references can never occur anywhere in top-level types.

TJVariable is used to refer to a type variable. It is identified by the *owner* of the corresponding type parameter, and by its index (Int) within the type parameter list of the owner. *TypeParameterOwner* is a simple enumeration type with two values:

- *TPOClass*: The type parameter belongs to a class declaration.
- *TPOMethod*: The type parameter belongs to a method declaration.

Index and owner are used to determine which types represent type parameters that need to be replaced with type arguments whenever such a substitution needs to be done. Finally, the last element of the *TJVariable* is the *TypeContext*: this is the full list of type parameters belonging to the owner. For each type parameter, name and upper bound are stored. This is used so that the upper bound of the type variable (which might be F-bounded) can be determined without having to pass a type context into all functions that work with types.

The constructor *TJFBoundRef* is used to solve the infinite recursion that would otherwise occur when representing the F-bounded type parameters with *TJVariable*. The *TJFBoundRef* is not considered a valid type and will occur only in the upper bound of type parameter declarations. Whenever the upper bound of a type parameter is accessed, all F-bound references are replaced with *TJVariable*s that correctly represent the upper bound as a type. The type parameter is identified using its index (similar to the first element in *TJVariable*).

TJCapturedWildcard is used to represent the fresh type variables created by capture conversion. An integer is used as unique identifier to tell apart the different types created by repeated invocation of wildcard capture on a parameterized type. Both lower and upper bound are stored with a captured wildcard type.

To represent intersection types, the type constructor *TJIntersection* is used. It is implemented as a simple list of types.

Finally, *TJNoType* is a special type used during elaboration that is used when type are not yet inferred, taking the role of **undef** in Section 4.2.

Using these definitions, the type operations defined in the type system chapter are implemented as Gofer functions.

For example, the implementation of type erasure follows directly from the definition in Section 3.8:

```
erasure :: JavaType > JavaType
erasure (TJP(t,_)) = TJRef t
erasure (TJArray e) = TJArray (erasure e)
erasure (TJIntersection (t:_)) = erasure t
erasure x@(TJVariable _) = erasure (upperBound x)
erasure t = t
```

6 Conclusions

6.1 Summary

We have introduced a type system for use with our ASM model. Based on the work of Daniel Smith [Smi07], we have built a type system that fully specifies the behavior of all Java types, including infinite types, which are left unexplained by the Java specification.

The major change when modeling Generic Java was the introduction of an explicit elaboration machine *elabJava* in order to model the static semantics of type checking and overload resolution. In some places, we simplified the algorithms by reusing definitions where the Java specification contains redundancies, for example when using applicability for defining the 'more specific' relation.

6.2 Observations

The vast changes to the type system in Java 1.5 made it problematic to model the changes as a separate sublanguage similar to the way the Jbook decomposes the Java language into five layers. Instead, we replaced existing definitions with extended versions of these definitions, or, in case of the overload resolution algorithm, reformulated the whole algorithm to support the new features. While Java Generics were designed to be backward-compatible, modeling the addition of Generics as conservative extension is problematic because the backward compatibility of Java is not achieved through existing language mechanisms, but through the careful addition of special cases (for example, using multiple passes in overload resolution) and compatibility constructs such as raw types.

Limitations in the Java specification have shown to be a bigger problem than expected.

An important observation is that a proof of type safety, if it should show the safety of using generic types, depends on assigning runtime type information to objects. However, the exact semantics for this are not clear as no proposal for reification has been chosen so far.

6.3 Future Work

The results of this work can build a basis for a full model of Java semantics.

Several other language features are also missing from the model:

- Inner classes and anonymous classes

- Static Import
- Enums
- Enhanced `for`-Loop

These have little to do with the Java type system, but nevertheless are necessary for a full model of Java. Inner classes in particular require a major change to the name lookup algorithm.

If the reification problem is solved, it will be possible to use this model to prove the type safety of Java by extending the proof from the Jbook.

Bibliography

[BÖ3] Egon Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15:237–257, 2003. 4

[BB08] Don S. Batory and Egon Börger. Modularizing Theorems for Software Product Lines: The Jbook Case Study. *J. UCS*, 14(12):2059–2082, 2008. 4

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM. 8

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003. 3, 4

[CDE08] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 2–26, Berlin, Heidelberg, 2008. Springer-Verlag. 9, 14, 20, 37

[CV08] Maurizio Cimadamore and Mirko Viroli. On the reification of Java wildcards. *Sci. Comput. Program.*, 73(2-3):59–75, 2008. 37, 38

[Ges] Júlio Vilmar Gesser. JavaParser: Java 1.5 Parser and AST. Web pages at http://code.google.com/p/javaparser/. 39

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, 3rd Edition*. Addison-Wesley Professional, 2005. xii, 1, 11, 13, 18, 22, 23, 25, 28

[IN06] Atsushi Igarashi and Hideshi Nagira. Union Types for Object-Oriented Programming. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1435–1441, New York, NY, USA, 2006. ACM. 14

[IV02] Atsushi Igarashi and Mirko Viroli. On Variance-Based Subtyping for Parametric Types. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 441–469, London, UK, 2002. Springer-Verlag. 8, 10

[KP06] Andrew J. Kennedy and Benjamin C. Pierce. On Decidability of Nominal Subtyping with Variance, September 2006. FOOL-WOOD '07. 25

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. xii, 8

[Mar] Simon Marlov. Happy: The Parser Generator for Haskell. Web pages at http://haskell.org/happy/. 39

[SC08] Daniel Smith and Robert Cartwright. Java Type Inference Is Broken: Can We Fix It? In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 505–524, New York, NY, USA, 2008. ACM. 14, 23, 25, 26

[Sch01] Joachim Schmid. Introduction to AsmGofer. Archived web pages at: http://web.archive.org/web/20070701135506/www.tydo.de/AsmGofer, 2001. 39

[Smi07] Daniel Smith. Completing the Java Type System. Master's thesis, Rice University, 2007. 1, 14, 15, 21, 25, 26, 32, 42

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. vii, xii, 1, 4, 19, 27, 28, 30, 34, 38

[TEH05] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL)*, 2005. 9, 14

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2007-03

2007-03	B. Bügling, M. Krosche	Coupling the CTL and MATLAB
2007-04	C. Knieke, M. Huhn	Executable Requirements Specification: An Extension for UML 2 Activity Diagrams
2008-01	T. Klein, B. Rumpe (Hrsg.)	Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, Tagungsband
2008-02	H. Giese, M. Huhn, U. Nickel, B. Schätz (Hrsg.)	Tagungsband des Dagstuhl-Workshopss MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV
2008-03	R. van Glabbeek, U. Goltz, J.-W. Schicke	Symmetric and Asymmetric Asynchronous Interaction
2008-04	R. van Glabbeek, U. Goltz, J.-W. Schicke	On Synchronous and Asynchronous Interaction in Distributed Systems
2008-05	M. V. Cengarle, H. Grönniger B. Rumpe	System Model Semantics of Class Diagrams
2008-06	M. Broy, M. V. Cengarle, H. Grönniger B. Rumpe	Modular Description of a Comprehensive Semantics Model for the UML (Version 2.0)
2008-07	C. Basarke, C. Berger, K. Berger, K. Cornelsen, M. Doering J. Effertz, T. Form, T. Gülke, F. Graefe, P. Hecker, K. Homeier F. Klose, C. Lipski, M. Magnor, J. Morgenroth, T. Nothdurft, S. Ohl, F. Rauskolb, B. Rumpe, W. Schumacher, J. Wille, L. Wolf	2007 DARPA Urban Challenge Team CarOLO - Technical Paper
2008-08	B. Rosic	A Review of the Computational Stochastic Elastoplasticity
2008-09	B. N. Khoromskij, A. Litvinenko, H. G. Matthies	Application of Hierarchical Matrices for Computing the Karhunen-Loeve Expansion
2008-10	M. V. Cengarle, H. Grönniger B. Rumpe	System Model Semantics of Statecharts
2009-01	H. Giese, M. Huhn, U. Nickel, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme V
2009-02	D. Jürgens	Survey on Software Engineering for Scientific Applications: Reuseable Software, Grid Computing and Application
2009-03	O. Pajonk	Overview of System Identification with Focus on Inverse Modeling
2009-04	B. Sun, M. Lochau, P. Huhn, U. Goltz	Parameter Optimization of an Engine Control Unit using Genetic Algorithms
2009-05	A. Rausch, U. Goltz, G. Engels, M. Goedicke, R. Reussner	LaZuSo 2009: 1. Workshop f—r langlebige und zukunftsfähige Softwaresysteme 2009
2009-06	T. Müller, M. Lochau, S. Detering, F. Saust, H. Garbers, L. Märtin, T. Form, U. Goltz	Umsetzung eines modellbasierten durchgängigen Enwicklungsprozesses für AUTOSAR-Systeme mit integrierter Qualitätssicherung
2009-07	M. Huhn, C. Knieke	Semantic Foundation and Validation of Live Activity Diagrams
2010-01	A. Litvinenko and H. G. Matthies	Sparse data formats and efficient numerical methods for uncertainties quantification in numerical aerodynamics
2010-02	D. Grunwald, M. Lochau, E. Börger, U. Goltz	An Abstract State Machine Model for the Generic Java Type System