`Rupiah`: An extension to Java supporting match-bounded parametric polymorphism, `ThisType`, and exact typing

by

Jonathan L. Burstein

A Thesis
Submitted in partial fulfillment of
of the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts
May 25, 1998

Abstract

The programming language Java [AG96] has gained widespread acceptance throughout the computer industry. Java's type system, though, is lacking in flexibility. This lack of flexibility limits the expressiveness of the language, especially for the creation of container classes.

To improve Java's expressiveness, we extend its type system through the addition of three constructs: match-bounded parametric polymorphism, `ThisType`, and exact typing. These constructs allow a programmer to write flexible, extensible, and statically type-safe code.

Our current implementation targets the standard Java Virtual Machine through a source-level translation. Translation allows `Rupiah` programs to be run on existing Java installations, but carries with it a performance cost.

We conclude with a comparison of our language changes and implementation and other proposals for extended Java.

Contents

List of Figures

Chapter 1

Introduction

In the past decade, programming languages that provide object-oriented features have become the tool of choice for the construction of complex applications and systems. When introduced, the facilities for data abstraction and code reuse that object-orientation provides were heralded as the innovation that would finally make programming a relatively quick and simple task. No longer would projects consistently fail to meet deadlines or the needed level of reliability, the scourges of programming in the large that have plagued the industry since its genesis.

Unfortunately, like so many other warmly received advances in computer technology, object-oriented languages have not lived up to their initial hype; programming remains a difficult, time-consuming, and error-prone task. In part, this is because programming is inherently difficult, time-consuming, and error-prone—nothing in the foreseeable (and perhaps unforeseeable) future will change this. Recognizing this fact, though, there is still room for improvement in the languages and tools that programmers use.

Statically-typed object-oriented languages such as C^{++} [ES90] and Eiffel [Mey92] provide strong features for data abstraction. Neither, though, fully succeeds in allowing for code reuse due to problems with their type systems. The type system of C^{++} is too strict, preventing code reuse in certain situations. The type system of Eiffel, on the other hand, permits certain unsafe practices—it is possible to write an Eiffel program that will pass all compile-time type-checking, but will fail with a type error when run.

The latest programming language to be hyped as the savior of programmers is Java [AG96]. The main attraction of Java is that instead of targeting a specific architecture and operating system (*e.g.*, a personal computer with an Intel processor running Microsoft Windows), a Java compiler targets a *virtual machine*. Any computer that provides the Java Virtual Machine (JVM) [LY97] can run any Java program. Furthermore, Java contains strong security and safety features, preventing programmers from making certain common, and often difficult to detect, errors.

Java's type system, though, does not meet the goal of providing safe and flexible code reuse. The primary focus of this thesis, then, is to present a modification of Java's type system that increases that flexibility and expressiveness of the language without sacrificing safety. Specifically, we incorporate three concepts from the language $\mathcal{LOOM}$ [BFP97]: match-bounded parametric polymorphism, *ThisType*[1] and exact types. These changes do not alter the semantics of existing Java programs—backwards compatibility is maintained.

A further focus of the thesis is implementing the above modifications without destroying Java's cross-platform nature. Goals of the ideal implementation include:

- No changes to the Java Virtual Machine.
- Primitive types (*e.g.*, `int`) can be used as type parameters.
- Minimal performance costs.
- Interoperability with existing Java classes and libraries.
- Security equivalent to existing Java implementations.
- Reasonable interaction with Java arrays.
- No needless duplication of code.
- Parameterized types should be equivalent to conventional types.
  - Support for the `instanceof` operator.
  - Checked casts.
  - Dynamic allocation using the `new` operator.
- Avoidance of name mangling.

Unfortunately, these goals are not mutually compatible. For example, it is not possible to use primitive types as type parameters without changing the JVM or resorting to name mangling and needless duplication of code. Also, maintaining the Java security model requires expensive run-time checks.

1.1 Outline of the thesis

This thesis is divided into three sections. The first section presents background information. In Chapter 2, we discuss the various problems that statically-typed object-oriented languages face. In Chapter 3, we give a brief description of the programming language Java

[1] *ThisType* is called *MyType* in $\mathcal{LOOM}$.

and examine the flaws of its type system. In Chapter 4, we summarize other proposals for extending Java, looking at the proposed language designs and implementations.

The second section describes the specifics of `Rupiah`, our extended version of Java. In Chapter 5, we describe our additions to the Java type system: `ThisType`, exact types, and match-bounded parametric polymorphism. In Chapter 6, we describe our implementation of `Rupiah`.

The last section evaluates the language design and implementation of `Rupiah`, and compares our work to the other proposals for extensions to Java. In Chapter 7, we evaluate and compare the various language designs (*i.e.*, the changes to Java's syntax and semantics). In Chapter 8, we evaluate and compare the different implementation strategies for extended versions of Java. In Chapter 9, we describe future avenues of research. Finally, the appendices contain sample `Rupiah` programs, and a grammar of `Rupiah`'s syntax.

Chapter 2

Problems with Static Object-Oriented Type Systems

In this chapter we present a brief and informal overview of type systems for object-oriented languages, focusing on the qualities that programmers desire of such systems, and the problems that those desires can cause. A more detailed exploration of these issues can be found in [Bru97c]. We assume the reader is already familiar with the basic concepts of object-oriented programming and the features that object-oriented languages provide to support those concepts, such as encapsulation and inheritance. This chapter is concerned mainly with current problems in object-oriented type systems; we present one possible solution in Chapter 5.

2.1 Safety and Flexibility

The primary goal of a type system is to provide safety—to detect dangerous or nonsensical operations, such as dividing the string "yellow" by the integer 17, before they are executed. By associating every value the program generates with a type, it is possible to check for such unwanted conditions. These checks can happen while the program is running—dynamic typing—or when the program is compiled—static typing. (As it is not always possible to completely check a program at compile time, though, many statically typed languages will defer some checks to either link or run time, depending on when the needed information will be available.)

Static type checking provides a number of advantages over dynamic checking. First, static type-checkers can detect more errors earlier in the programming process; while a dynamic type-checker can only detect one error at a time (as the program terminates once an error is detected), a static type-checker can detect multiple, and hopefully (though not always) provide more informative error messages to the programmer. Furthermore, statically typed languages are often more efficient than dynamically typed languages in terms of both time and space, as no code needs to be included to perform run-time checks, type information does not need to be kept in memory during execution,[1] and modern compilers can use the compile-time type information to perform better code optimization. Finally, the declarations that most statically typed languages require improve the readability of the source code.

However, a statically typed language can also be less expressive (and thus more difficult to use) than a dynamically checked language. Because a static type-checker must be conservative, it might reject a program that will in fact execute without error. In describing Pascal [Wir71], for example, Bjarne Stroustrup stated "I had found [its] type system worse than useless—a straitjacket that caused more problems than it solved..." [Str94]. A programmer can get around the restrictions that a static type-checker might impose, but doing so will normally result in a loss of performance, safety, or clarity—the exact things that static type-checking was to have provided.

Object-oriented languages have more complex type systems than procedural or functional languages due to the presence of subtyping and the need for an object to be able to refer to itself (normally, through a *pseudo-variable* such as `self` or `this`) when executing a method. The primary focus of object-oriented type systems is determining whether or not a message can be sent to a particular object (*i.e.*, determining if the receiver has a method that can execute in response to the message). Subtyping and inheritance result in situations where the type of an object is different at run-time than compile time. A static object-oriented type system must be able to safely support these features (and therefore allow for the reuse of code, one of the main reasons for object-oriented programming) without limiting their flexibility, or relying on run-time checks.

2.1.1 Inflexibility in current OO languages

Unfortunately, many current object-oriented programming languages, such as C^{++} [ES90], Java, Object Pascal [Tes85], and Modula-3 [CDG+88], possess type systems that place too many restrictions on flexibility. To get around these restrictions, the programmer must resort to using type casts in C^{++}, Object Pascal, and Java (which are checked at run-time only for Java) or the "typecase" statement of Modula-3 (which depends on run-time type information). The main flaw in these languages' type systems is that they do not allow changes in the types of methods (*i.e.*, the types of parameters or the return type) in subclasses.[2] The following example illustrates why this is problematic.

Figure 2.1 shows code segments from two classes: `Circle` and `ColorCircle`. The first

[1] For the purposes of type checking; run-time type information might still be needed for other purposes, such as for a type-case statement.

[2] The new C^{++}standard allows for the return type of a method to be replaced by a subtype in a subclass, as long as that return type is a pointer type.

```
class Circle {
  private Point center;
  private float radius;

  public Circle(Point center, float radius) {...}
  public Circle clone() {return new Circle(center, radius);}
  public Point getCenter() {return center;}
  public void changeCenter(Point newCenter) {center = newCenter;}
  public void move(int dx, int dy) {...}
}

class ColorCircle extends Circle {
  private Color color;

  public ColorCircle(ColorPoint center, float radius) {...}

  public Circle clone() { // Actually returns a ColorCircle
    return new ColorCircle(new ColorPoint(center, color), radius);
  }

  public Point getCenter { // Actually returns a ColorPoint
    return new ColorPoint(center, color);
  }

  public void changeCenter(Point newCenter) { // Should take a ColorPoint?
    center = newCenter;
    // color = newCenter.getColor()?
  }
}
```

Figure 2.1: Changing types in inherited methods

method of interest is `clone()`, which returns a copy of the object; in `Circle`, it returns a new object of type `Circle`.[3] In `ColorCircle`, `clone` actually returns a `ColorCircle`, but because a subclass is not permitted to change the method signature, `clone()` is still declared as returning a `Circle`. A programmer would need to cast the returned object to `ColorCircle`. The method `getCenter()` has the same problem: in `ColorCircle`, it is declared as returning a `Point`, even though it actually returns a `ColorPoint`. For both of these methods, we should be able to change the return type, but the type system does not let us.

The method `changeCenter()` presents an interesting problem: should we be able to change the type of the parameter so that in `ColorCircle` it takes a `ColorPoint` instead of a `Point`?

2.1.2 Changing method signatures in subclasses

The example in Figure 2.1 shows the need for being able to change the signature of an inherited method in a subclass. There are two cases that we need to consider: changing a method's return type, and changing the type of one of the method's parameters. Because a method might call other methods within the class, though, we must ensure that any changes made in a subclass will not cause a problem if that overridden method is called by a method in the superclass (*i.e.*, a method that is inherited but not changed).

The rule for a return type is simple and intuitive: in a subclass, we can change a method's return type to a subtype of the original return type. This is referred to as a *covariant* change; the return type and the class both change in the same direction. Such a change is safe, as a subtype can always be used in place of a supertype; a method that expected to receive a `Point` when calling `getCenter()` can receive a `ColorPoint` without any difficulty.

The covariant typing rule does not work for the types of parameters, however. Imagine that in a program, a variable c with a static type of `Circle` actually referred to a `ColorCircle`. Since `ColorCircle` extends `Circle`, it is a subtype of `Circle`; an object of type `ColorCircle` can therefore be assigned to a reference of type `Circle`.[4] The program calls c.`changeCenter()` with an actual parameter of type `Point`, even though `changeCenter()` in `ColorCircle` has been changed to take a `ColorPoint`—a type error has occurred. In a subclass that is to be used as a subtype, the types of parameters cannot change covariantly.

Parameter types can change *contravariantly*, though; in a subtype, it is safe to change a parameter's type to a supertype. Continuing the circle example, it would be perfectly safe to change `changeCenter()` to take a parameter of type `Object`. While such a change is correct from a type perspective, it is rarely useful from a programming perspective. The method

[3] In Java, `clone()` is defined in `java.lang.Object`, the root class, so it actually returns `Object`.

[4] Subclasses need not always generate subtypes, though; see Section 2.2.

`changeCenter()` needs the information a `Point` provides to perform its job; changing the parameter of `changeCenter()` to be of type `Object` in `ColorPoint` is type correct, but not semantically useful.

Eiffel: Trading safety for flexibility

In Eiffel [Mey92], covariant changes are allowed in subclasses, even though such changes are not type-safe. The designers of the language felt that flexibility of allowing method parameter type changes outweighed the loss of safety, and that run-time errors from such changes would not occur very frequently. Others, of course, felt that allowing covariant changes was unsound and therefore should be avoided, even though such changes improve the flexibility of the language.

Both views are correct; the problem is that the user is forced to make a choice between flexibility and safety. In Section 5.1, we present a solution to this problem—a type system that allows safe covariant changes to the types of a method's parameters for many cases, including those of Section 2.2.

2.1.3 Changing the types of instance variables in subclasses

An instance variable can be read from and assigned to. A single instance variable, then, can be viewed as actually being two methods: for an instance variable x of type T, we would have the methods

```
T getX();
void setX(T newX);
```

We can therefore consider the type of an instance variable to be both a return type and a parameter type.

Because of this, it is clear that by the above rules, we cannot change the type of an instance variable. If an instance variable's type could change covariantly, assignment to that variable would become dangerous. If an instance variable's type could change contravariantly, using the value of that variable would become dangerous. Figure 2.2 gives examples of both of these kind of changes. (Note that this is true for all instance variables except those that are private; that is, public, protected, and "default access" instance variables. As subclasses do not inherit private instance variables, it would be impossible to change their type.)

There are times, though, when we would desire to be able to change the type of an instance variable. In a singly-linked node, for example, the variable `next` has type `SingleNode`. In a doubly-linked node, `next` should have type `DoubleNode`. Not having the ability to alter the types of instance variables thus makes it difficult to write `DoubleNode` as a subclass of `SingleNode`.

```
class String extends Object { ... }
class UpperCaseString extends String { ... }

class C {
  public String s;
  public String ss;
  ...
}

class D extends C {
  public UpperCaseString s;   // Covariant change
  public Object ss;           // Contravariant change
  ...
}

C c = new D();
c.s = new String(``A string'');  // Error: s is an UpperCaseString
String a = c.ss;                 // Error: ss is an Object
```

Figure 2.2: Changing the type of an instance variable

```
class C {
  ...

  public boolean equals(Object anObject) {
    if (anObject instanceof C) {
      ... // the actual check for equality
    } else {
      return false;
    }
  }
}
```

Figure 2.3: Binary methods and Java

2.2 Binary methods: Why subclasses aren't always subtypes

A *binary method* is a method that takes a parameter of the same type as the class in which the method is defined; a common example is an `equals()` method. Languages that do not allow method signatures to change in subclasses, though, do not handle binary methods well. In Java, for example, `equals()` is defined in `java.lang.Object`, so it takes a single parameter of type `Object`. Classes that modify `equals()` often have code similar to Figure 2.3—the method must check to make sure that the object it is passed is of the correct type before actually checking for equality. Were changes in method signatures allowed, this type check would not be necessary.

It would seem, then, that Eiffel is justified from a programming perspective in allowing covariant changes to parameters' types in subclasses, even though such changes are type-unsafe. There is a solution to this problem (see Section 5.2.2); however, the solution requires a major change in the way people think about object-oriented type systems. The key insight is that when a class contains a binary method, a subclass that changes the type of the parameter in the binary method cannot safely be used as a subtype.

For a more detailed examination of binary methods in object-oriented languages, see [BCC$^+$95].

2.3 Real-world data structures: Parametric polymorphism

A common data structure is a *container*: a structure, such as a list or a set, that is meant to hold other structures. Because they are used so frequently, containers can obviously benefit from the potential for reuse that object-oriented languages provide; a programmer should only need to write a specific container class once, for the code that manipulates the

```
class List {
    public void insertHead(Object o) {...}
    public Object getHead() {...}
}

List sl = new List();  // A list of string

sl.insertHead(new String("Hello"));
String h = (String)sl.getHead();    // Programmer must insert cast

sl.insertHead(new Object());        // Allowed by the compiler
String o = (String)sl.getHead();    // Cast will fail at runtime
```

Figure 2.4: Simulating parametric polymorphism

container remains the same no matter what type of object is being held. An inflexible type system, however, can make container classes much more difficult for a programmer to use than they need be.

The simple solution for implementing container classes in an object-oriented language is to parameterize the container, so that the programmer can choose what type of objects the container is to hold, and the compiler can ensure that only objects of the correct type are inserted into the container. This is known as *parametric polymorphism*. A programmer, for example, could create a `List<String>`—a list of strings—or a `List<List<String>>`—a list of lists of strings; the class `List` would only need to be written once.

A programmer can simulate parametric polymorphism in a language that does not support it by making use of subtype polymorphism, the ability of a reference to an object of a given type to also refer to objects that are subtypes of that type. This carries two penalties: a cast is required when removing an object from the container, and the compiler cannot ensure that only objects of the correct type are being inserted into the container. See Figure 2.4 for an example.

2.3.1 Bounded parametric polymorphism

Parametric polymorphism is not enough, though, for a container class might not be able to hold any arbitrary type. An ordered list, for example, can only hold classes that provide a method that determines order, such as `lessThan()`. There needs to be a way to ensure that only classes that provide the required methods are used as a parameter to the container class.

The solution is *bounded parametric polymorphism*: when defining a container class, the programmer includes conditions that a class meant to be used as a parameter must meet. The compiler can check these bounds, and raise an error if a class given as a parameter does not meet them. The container class need only be type-checked once.

As the bounds often involve binary methods, though, designing a type system that handles bounded parametric polymorphism in a safe and flexible manner is not a simple task. Section 5.3 describes one possible solution.

Templates in C^{++}

C^{++}, through its template mechanism, provides an alternative solution to bounded parametric polymorphism. Because the parameter to a container class is unbounded, that compiler must type-check the class for every set of type parameters used to instantiate the class; for example, the class `OrderedList` must be checked when an `OrderedList<String>` is created, and again when an `OrderedList<List<String>>` is created, with an error being raised for the latter.

While this is a usable alternative, it causes two problems. First, the compiler must have available the information needed to type-check the container class whenever a programmer wants to use that class. This complicates separate compilation and hampers modifying container classes. In C^{++}, the source code for template classes must be distributed to everyone who wishes to use them. Second, the programmer loses the documentation that bounds provide; he or she cannot tell what classes can be used as a parameter for a container class without examining the source code (or, hopefully, documentation provided by the programmer of the template class). Bounded parametric polymorphism does not have either of these problems. Because the bounds are part of the container class's interface, both the compiler and the programmer know what classes can be used to instantiate the container without looking at the source code.

Chapter 3

The Java Programming Language

Java has attracted more attention in a shorter period of time than any previous programming language, and most computer products in general. In part, this is because Java has been marketed[1] as a programming language, platform, operating system, moral crusade, and religion. In less than five years, Java has brought forth millions of venture capital dollars, calls for a revolution in how people use computers, lawsuits and counter-suits, and an endless stream of products that feel compelled to relate their name to coffee.

In this chapter, we present the basics of Java as a mere programming language: the features that have led to the hype (Section 3.1), and the limitations of its type system that are the reason for this thesis (Section 3.2).

3.1 Benefits of Java

The benefits of Java fall into two categories: it provides features that make it a good object-oriented language, and features that make it a strong choice for use in a networked world.

3.1.1 As a programming language

"Pure" Object-orientation

In Java, everything is an object, with the exception of the base numeric and character types (*i.e.*, `byte`, `short`, `int`, `long`, `float`, `double`, and `char`). This presents the programmer with a cleaner set of tools for building data structures than languages which are object-oriented extensions to procedural languages, such as $C^{++}$and Object Pascal. Because there

[1] Java has certainly been marketed more than any other programming language, with the advertising program including such things as commercials during NFL football games.

is only one tool for abstraction, the programmer does not need to learn the difference between a `struct`, `class`, and `union`, for example.

A programmer can choose to work in a non-object-oriented style, if desired, without incurring an excessive overhead cost. `Static` methods, for example, do not require an object instance, and thus can be used to program in a procedural style. Similarly, a programmer can make use of `final` classes and methods to provide data abstraction without the extensibility of objects.

Interfaces

Java only allows single inheritance for classes: all classes inherit methods from one, and only one, class (except for `Object`, the root of the class hierarchy). This avoids problems that multiple inheritance of methods can create, such as a class inheriting two different implementations of the same method.

However, as it is useful for a class (an implementation of methods) to exist in different places in the type hierarchy, Java provides interfaces. An `interface` is a collection of method signatures without implementation—`abstract` methods (or, in C^{++} terminology, *pure virtual* methods). A class, then, inherits methods from another class, and can choose to implement the abstract methods of any number of interfaces.

Because of interfaces and inheritance, a single class has multiple types: the type of the class, the type of the class it `extends` (and thus recursively all of the classes on the inheritance chain between itself and `Object`), the type of any interface it `implements`, and all of the interfaces that those interfaces extend.

Interfaces allow a programmer to use multiple programming styles. A programmer, for instance, can define an interface for every type to be used by the program, separating method implementations from the object types. Alternatively, a programmer can use interfaces to provide some of the features of multiple inheritance, such as when an object needs to be used as types from different areas of the type hierarchy (*e.g.*, as a graphic interface component and as a thread).

Automatic Memory Management

In Java, all object instances are heap-allocated, and are manipulated through references (*i.e.*, a pointer to the object instance). An object instance is only allocated through the explicit use of the `new` operator.[2] A program may not directly reference memory; the only way to access the heap is through an object reference.

[2]Object instances may also be created dynamically through the `newInstance()` method of `java.lang.Class`. Dynamic object creation is needed to properly support the component programming paradigm (*e.g.*, JavaBeans). The `clone()` method also allocates new object instances.

Objects are garbage collected; the memory an object instance inhabits is reclaimed when the system determines that the object is no longer in use. The programmer cannot explicitly deallocate an object. Garbage collection prevents both memory leaks (not deallocating an object that can no longer be used) and the dangling reference problem (attempting to access an object instance that has been deallocated). As such bugs can be the most difficult for a programmer to track down, automatic memory management greatly improves programmer productivity and program reliability.

Concurrency

Java provides language level support for concurrent programming (multithreading). The language includes synchronization primitives, and every object instance has a lock associated with it. The standard Java class library (*i.e.*, the package `java.lang` and its sub-packages) provides for the creation, manipulation, and control of threads.

Class libraries

The Java Language Specification defines a large number of classes that must always be available to a program. These classes include such things as user interface components, standard data structures, and methods of accessing local or remote data. Other class libraries provide for distributed computing, database access, and many other common tasks.

3.1.2 In a networked world

Java is implemented in a way that makes it especially attractive for use over networks, especially the Internet. This attractiveness is the underlying reason for much of the hype around Java.

Architecture independence

The majority of Java compilers do not target a specific operating system or hardware architecture; instead, they output bytecodes for the Java Virtual Machine [LY97]. Any computer that possesses a Java Virtual Machine (JVM) implementation can execute Java bytecodes, and therefore can run a Java program.

Dynamic linking and Separate Compilation

The output of Java compilers, `.class` files, contain symbolic references—the names of methods and classes, for example. The references are not resolved until the code is executed. This delaying of the resolution of symbolic references (*i.e.*, linking) until loadtime makes extending and reusing code easier—a change in one class will not require other classes to be recompiled (as long as certain compatibility rules are met), and a new class can be added to an already running system.

Furthermore, because `.class` files contain detailed type information, a Java compiler does not need access to the source of classes that are referenced by the class that is being compiled. Unlike C or C^{++}, the executable code can be used to perform compile time type-checking; no header files or type libraries are needed.

Safety

Java uses a three-pronged approach to safety. First, the language does not allow most activities that might be unsafe, such as directly accessing memory, manipulating pointers, or deallocating objects. Potentially unsafe actions, such as type casts or array assignment (see Section 3.2.2), must be checked at runtime.

Second, the Java Virtual Machine verifies `.class` files before they are used.[3] This process ensures that a buggy or malevolent compiler has not created a `.class` file that does not conform to specification.

Finally, the JVM executes untrusted code—*i.e.*, code that was loaded from an unknown source—in a *sandbox*. Code running in the sandbox has very limited access to system resources, such as the local file system, and thus cannot damage the system or violate a user's privacy. Newer Java implementations allow for multiple levels of trust, allowing for fine-grained access to system resources.

3.2 Java's type-system flaws

While Java makes use of many recent advances in object-oriented programming language technology, its type system unfortunately does not. Java's type system is stricter than it needs to be, forcing the programmer to use workarounds such as type casts. Even worse, in an attempt to increase the flexibility of the type system, a static type hole was created for arrays, necessitating a run-time check.

3.2.1 Java's lack of flexibility

The main problem with Java's type system is that it is too inflexible; as with the languages described in Section 2.1.1, Java does not allow a subclass to alter the types of inherited methods. This lack of flexibility forces a programmer to make use of type casts or explicit type-checks, through the `instanceof` operator. For example, a programmer must cast the value returned from the `clone()` method to the proper type before using that value. In the `equals()` method, a programmer needs to check the type of the actual parameter, and

[3] The verification process can be skipped for code that is assumed to be safe.

possibly cast the parameter to the type of the object executing the method. A more flexible type system that incorporates the ideas presented in Chapter 5 would not require these workarounds.

Java also does not provide for parameterized types (parametric polymorphism). As an object of any type can be assigned to a reference of type `Object` (as `Object` is at the root of the Java type hierarchy), a main use of parameterized types—container classes—can be simulated by simply having the container store objects of type `Object`. This workaround is unsatisfactory, though. First, the programmer must explicitly cast objects to the proper type when removing them from the container; as casts are checked at run-time, this incurs a performance cost. Also, there is no way to define a homogeneous container (*i.e.*, one that only holds objects of a single type). As the compiler allows an object of any type to be inserted into a container, a programmer mistake could result in an unexpected object being placed into the container, and therefore a failed cast when that object is removed.

The above technique can also be used to simulate bounded parametric polymorphism. Instead of using `Object` for the types of values, a programmer could use a different class or interface type. Only objects that extended the class or implemented the interface could be placed into the structure. The programmer would still need to explicitly cast objects when removing them from the container.

3.2.2 Arrays

Bounded parametric polymorphism can be used to write a routine that operates on a value of any type that satisfies the bound; for example, one could write a generic sort routine that would take as input a type that has a `lessThan()` method, and an array of elements of that type. To provide a way to write such generic routines, the designers of Java explicitly introduced a static type hole (*i.e.*, a type error that can be detected only during program execution, not during compilation) into the language.

The array type rules in Java were designed so that this use of parametric polymorphism could be simulated. In Java, if ET is a class that extends T, an object of type ET can be assigned to a reference of type T. This is referred to as assignment compatibility.[4] If we have a reference to an array A_{et} of type $ET[]$, and a reference to an array A_t of type $T[]$, Java allows A_{et} to be assigned to A_t. This is not type safe, though. Because the element type of the arrays changes covariantly, an array of type $ET[]$ is not a subtype of an array of type $T[]$.

Figure 3.1 illustrates the danger of Java's array assignment rules. Because ET extends T, the use of aet (of type $ET[]$) as an actual parameter for `danger(T[] at)` is allowed. In `danger()`, the static type-checker allows an object of type T to be placed in the array at. If the assignment were performed, we would run into the problem shown in `broken()` after

[4]The assignment compatibility results because ET extends T, and is therefore treated as a subtype. Interestingly, the Java Language Specification never uses the term subtype.

```
class ET extends T {...}

public void danger(T[] at) {
  at[0] = new T(); // Must be checked at runtime
}

public void broken() {
  ET[] aet = new ET[5];

  danger(aet);
  aet[0].someMethodNotInT();  // An error would result here
}
```

Figure 3.1: Type problems with Java's arrays

the call to `danger()`—the program calls a method that is present in ET but not T. As $aet[0]$ is really an object of type T a method not found error would occur.

Because the designers of Java were well aware of this type hole, the Java specification mandates that a run-time check be performed whenever an object reference is stored into an array. This check must be performed for every store—a major performance hit, as most programs do not use arrays in a way that would ever cause the check to fail. The lack of flexibility in Java's type system resulted in a compromise that greatly impacts the performance of the language; a more flexible type system that would not require such a compromise would be both safer and faster.

Chapter 4

Other Extensions to Java

Here, we examine other proposed extensions to Java meant to improve the flexibility of the type system. Section 4.1 describes the proposals' changes to Java's type system, and Section 4.2 describes the implementation of these changes. We briefly evaluate both the proposed changes and implementation in Section 4.3.

4.1 Proposed type-system changes

4.1.1 F-bounded parametric polymorphism

Pizza [OW97], Generic Java [BOSW97], Refined Java [CJ98], and [AFM97] propose adding *F-bounded parametric polymorphism* [CCH$^+$89] to Java to improve the language's support for building and using container classes. Bounds for parameterized types are expressed through the use of Java's `extends` and `implements` relations. For example, if a class C is declared as

```
class C<T extends B>  // B is a class
```

any type P used as an actual parameter to C[1] must be a subtype of (*i.e.*, `extend`) B. Similarly, if a class D is declared as

```
class D<T implements I> // I is an interface
```

any type P used as an actual parameter must extend I, if P is an interface, or must implement I, if P is a class. (If P implements an interface J that extends I, then P also implements I.)

[1] Passing an actual parameter to a parametric class is called *instantiating* the class; the resulting type (*e.g.*, C<P>) is an *instantiation* of C.

```
interface HasLTandEquals<T> {
    boolean lessThan(T other);
    boolean equals(T other);
}
```

Figure 4.1: Binary methods and F-bounded parametric polymorphism

Interfaces may also be parameterized, and parameterized interfaces may be used as bounds. This is necessary for bounded parametric polymorphism to properly support binary methods (*i.e.*, a method that contains a parameter that is the type of the enclosing class). In an ordered list, for example, any type that is to be used as an element of the list must have a `lessThan()` and `equals()` method. To build a parametric ordered list, one would first define the interface `HasLTandEquals<T>` (see Figure 4.1). The ordered list would then be defined as:

```
class OrderedList<E implements HasLTandEquals<E>>
```

This bound means that a type ET used as an actual parameter to `OrderedList` must have the methods `boolean lessThan(ET other)` and `boolean equals(ET other)`, and thus can be ordered.

Additional language features

Pizza adds two other features to Java: higher-order functions and algebraic types. Higher-order functions allows for functions (methods) to be passed as arguments, returned as results, or stored in variables. Algebraic types are useful for creating structures such as abstract syntax trees, when there are a set number of node types, but a varying number of operations that one wishes to perform on those types. As both of these features can easily be written in standard Java through the use of inner classes and anonymous classes [Jav97] (though not as tersely), they will not be discussed further.

Unlike the other proposals, Pizza makes use of type inference in constructors. When creating a new instance of a parameterized class, the programmer does not need to include the actual type parameters in the call to the constructor. In Pizza, a programmer would write

```
List<String> aStringList;
aStringList = new List();  // <String> not needed
```

The other proposals require the programmer to write

```
aStringList = new List<String>();
```

```
class Name {
  int hashCode() { ... }
  ...
}

interface Hashable implementedBy Name {
  int hashCode();
}
```

Figure 4.2: Post-facto `implements` declaration

[AFM97] proposes adding an `implementedBy` construct to Java. This construct allows a programmer to create post-facto type relationships; for example, one can create a new interface and declare that older classes implement that interface. In Figure 4.2, for example, `Name` is declared to implement `Hashable` after the fact. This feature is useful when bounds need to be added to an existing library of classes; the needed type relationships can be declared without modifying the library.

4.1.2 Where clauses

An alternative method of bounding parametric polymorphism is given in [MBL97]. Bounds on a parameter are expressed through *where clauses*, a list of signatures of methods and constructors that objects of the actual parameter type must support. An ordered list definition, for example, would be written as:

```
class OrderedList[T] where T {boolean lessThan(T other);
                              boolean equals(T other); }
```

Only actual parameters that satisfy the where clause—*i.e.*, object types that possess methods compatible with the signatures in the where clause—can be used to instantiate the parameterized class. If there are multiple methods that might be used, due to overloading, the closest matching method is used.

The argument for using where clauses instead of F-bounded polymorphism is that it does not require a class to explicitly declare itself as being a subtype of a class or interface meant to be used as a bound. Adding a new parameterized class, with new bounds, would not require rewriting previously written types so that they implement the interface of the new bounds. Also, interfaces in Java cannot contain constructors or static methods, so where clauses are more flexible.

4.1.3 Virtual types

Instead of adding bounded parametric polymorphism, [Tho97] proposes adding *virtual types*, found in the language Beta [KMMPN87], to Java. Virtual types allow a user to redefine a type name in a subclass. For example, a generic container would be written as

```
class Container {
  typedef ElemType as Object;

  void addElement(ElemType e) {...}
  ...
}
```

To create a specialized container, a programmer would extend a class containing virtual types as follows:

```
class StringContainer extends Container {
  typedef ElemType as Point;
}
```

It is not possible to only type-check classes using virtual types at compile time, as the redefinition of a virtual type in a subclass can result in a covariant change to a parameter. In the above example, `Container` contains the method `addElement(Object e)` (substituting `Object` for the virtual type `ElemType`), while `StringContainer` has the method `addElement(String e)`. Figure 4.3 shows how this change can result in a type error that can only be detected at runtime: the call to `addElement()` will fail because an `Object` is being passed to a method that expects a `String` as a parameter.

Virtual types are very useful for writing and extending mutually recursive classes, which are often found in standard design patterns. Current statically typed object-oriented languages do not allow for mutually recursive classes to be extended in a easy manner. However, [Bru97b] and [BOW98] show how this can be done in a statically typed language. Furthermore, virtual types are a major change to the type system of Java, and one most programmers are not familiar with.

4.2 Proposed implementations

In evaluating extensions to Java, the implementation is as important as the details of the extension itself. Ideally, an extension should work seamlessly with the large number of current Java Virtual Machine implementations, and should not alter the operation of existing Java programs, either in source or bytecode form. Inertia, at the very least, will prevent an

```
class Container {
  typedef ElemType as Object;
  void addElement(ElemType e) {...}
}

class StringContainer extends Container {
  typedef ElemType as Point;
}

Container c = new StringContainer();
c.addElement(new Object());          // A runtime error occurs here
```

Figure 4.3: Virtual types and type safety

extension that depends on changes to the JVM from gaining widespread acceptance, even if the extension is far superior to what is currently in use.[2]

4.2.1 Homogeneous translation

Pizza and Generic Java are implemented by translating the source code into standard Java source, and then compiling that source into JVM bytecodes.[3] [OW97] points out that there are two methods of performing the translation to Java: one can create a new Java class for each instantiation of a parameterized class (*heterogeneous* translation), or one can translate the parameterized class into a single Java class (*homogeneous* translation). Heterogeneous translation is equivalent to macro expansion, and is similar to most implementations of templates in C^{++}. See Figure 4.4 for an example of the two translation methods.

The current Pizza implementation performs homogeneous translations. It does not, though, store any extra type information (*e.g.*, the type used to instantiate the object) for use at runtime. The rationale for not storing extra type information is that it allows Java library classes, such as `java.util.Vector`, to be instantiated as if there were parameterized classes. The programmer is able to gain from the improved type system while leveraging an already existing code base.

Not including the extra type information causes many problems, however. First, casts involving parameterized types cannot be checked properly; an invalid cast will not be de-

[2]This is a concern that must not be underestimated. A major reason for the success of $C^{++}$was its ability to interoperate cleanly with the incredible amount of C code that was already in use, and common development tools, such as linkers.

[3]The Pizza compiler performs the translation at the level of its internal representation; actual Java source code is generated only if the programmer wishes to see it.

```
// The parameterized class
public class Item<T> {
  T theItem;

  public Item(T item) {
    theItem = item;
  }

  public T getItem() {
    return theItem;
  }
}

// Using the translations
Item<String> strItemm;
String s;

strItem = new Item(``Hello'');


// Heterogeneous translation
s = strItem.getItem();

// Homogeneous translation
s = (String)strItem.getItem();
```

```
// The homogeneous translation
public class Item {
  Object theItem;

   public Item(Object item) {
     theItem = item;
   }

   public Object getItem() {
     return theItem;
   }
}

// The heterogeneous translation
public class Item_String {
  String theItem;

  public Item_String(String item) {
    theItem = item;
  }

  public String getItem() {
    return theItem;
  }
}
```

Figure 4.4: Homogeneous and Heterogeneous translations in Pizza

tected until an item is removed from the container. Similarly, the `instanceof` operator cannot be used with parameterized types.

Arrays are also problematic for the Pizza and Generic Java implementation. In a parameterized class `C<T extends Object>`, constructing a new array of type T would be translated into an array of type `Object`:

```
class C<T extends Object> {    // Original version
  T[] getNewArray(int size) {
     return new T[size];
  }
}

class C {                      // Translated version
  Object[] getNewArray(int size) {
     return new Object[size];
  }
}
```

If C is instantiated with any type except for `Object`, any call to `getNewArray()` will fail, due to an illegal cast. If `String` were used as the instantiation of C, for example, a call to `getNewArray()` would be translated as

```
C<String> cs = new C();
String[] sa = (String[])cs.getNewArray(5);
```

However, since `getNewArray()` returns an array of Objects (*i.e.*, an array created with type Object[]), the cast to an array of String will fail. The problem here is with how the array was created within the parameterized class, and not the signature of the method; if `getNewArray()` created an array of String in the above example, the cast would succeed.

Finally, it is possible for a hostile compiler or a person using multiple casts to insert an object of an improper type into a Pizza or Generic Java collection class. As pointed out in [AFM97], this can result in a security flaw. A programmer who is aware of this flaw in the translation scheme can avoid it by explicitly checking types if security is an issue; however, the point of extending Java is to avoid having the programmer expend effort to work around flaws.

4.2.2 Mixed translation

The implementation of Refined Java [CJ98] is meant to correct the flaws of Pizza and Generic Java's homogeneous translations. The Refined Java implementation combines the concepts of homogeneous and heterogeneous translation. A parametric class compiles into a single `.class` file, and each distinct instantiation of that class results in two additional `.class` files.

Refined Java translates parameterized classes in a similar manner to the homogeneous translations of Pizza and Generic Java. Within the body of the class, type variables are replaced by their bounds. Unbounded type variables are replaced by `Object`. This class is known as the *base class*. Compiling the class `Vector<T>` results in the base class `Vector`.

In a client class (*i.e.*, a class that makes use of a parameterized class or interface), instantiated types for variables are replaced with the type of the base class. A variable of type `Vector<String>`, for example, is translated to type `Vector`.

Each distinct instantiation of a parameterized class results in the construction of a new class and interface. The new class, termed a *wrapper class*, extends the base class. The name of the wrapper class encodes the actual type parameters for the instantiation. For example, the instantiation `Vector<String>` results in the wrapper class `$$Vector$_String_$`, which extends the base class `Vector`. The constructors of the wrapper class simply call the superclass version of the constructor (*e.g.*, `super(...)`).

In the translation of a client class, constructor calls for instantiated types are translated to constructor calls for the appropriate wrapper class. For example,

```
Vector<String> vs = new Vector<String>(5);
```

translates to

```
Vector vs = new $$Vector$_String_$(5);
```

Note that the wrapper class is only used for the constructor; the variable's type is of the base class, not the wrapper class.

The generation of wrapper classes alters the type hierarchy, however. For example, if `Stack<T>` extended `Vector<T>`, then `Stack<String>` should extend (*i.e.*, be a subtype of) `Vector<String>`. The above translations results in the following hierarchy:

```
class Vector;
class $$Vector$_String_$ extends Vector;
class Stack extends Vector;
class $$Stack$_String_$ extends Stack;
```

The translation does not maintain the desired subtype relationship. To capture this relationship, Refined Java adds *wrapper interfaces*. Wrapper interfaces encode actual type parameters, but do not contain any methods. The inheritance hierarchy of the wrapper interfaces exactly matches that of the instantiated types. For the above example, the compiler generates the following wrapper interfaces:

```
interface $Vector$_String_$;
interface $Stack$_String_$ extends $Vector$_String_$;
```

A wrapper class implements the corresponding wrapper interface.

Wrapper interfaces are used to implement casts and `instanceof` operation involving instantiated types. For example,

```
if (x instanceof Vector<String>)
...
Vector<String> vs = (Vector<String>)x;
```

translates to

```
if (x instanceof $Vector$_String_$)
...
Vector vs = (Vector)(($Vector$_String_$)x);
```

Because the wrapper classes are empty, two casts are required for the translation of a cast to an instantiated type. The cast to the wrapper interface checks that the value is of the proper instantiated type (*i.e.*, that it has the correct actual type parameters). The cast to the base class is needed to allow the value to be used.

Operations mentioning type variables

Within the body of a parametric class, operations involving a type variable, such as array allocation, require further translation. Type erasure (*i.e.*, replacing a type variable with its bound) will not suffice, as the runtime value of the type variable is needed.

Refined Java supports these operations through the use of *snippets*. A snippet is a compiler-constructed method that performs the necessary operation. In the base case, the snippet performs the operation using the type of the bound. In a wrapper class, the snippet method is redefined to use the correct value for the type parameter. For example, within a parameterized class that has an unbounded type variable `T`, `new T[x]` would translate to `$snip$1(x)`, where `$snip$1()` is a method that returns an array of `Object`.[4]

Figures 4.5 and 4.6 give an example of Refined Java translations. Figure 4.5 shows the original class, and the base class translations. Note that the array allocation expression (`new T[size]`) in the constructor is replaced in the translated version with a call to the method `$snip$1(size)`; this method performs the actual allocation, and returns a reference to the new array. Figure 4.6 shows the wrapper class translation for a specific instantiation. Note that the wrapper class implements the corresponding wrapper interface, defines a constructor to call the superclass (*i.e.*, base class) constructor, and redefines `$snip$1` to allocate an array of the proper type.

[4]There are additional complications regarding the accessibility of actual type parameters which we will not discuss. See [CJ98] for a description of these cases.

```
class Vector<T> {
  T[] elements;

  public Vector(int size) {
    elements = new T[size];
  }

  public T getElement(int pos) {
    return elements[pos];
  }
  ...
}

// Refined Java base class
class Vector {
  Object[] elements;

  public Vector(int size) {
    elements = $snip$1(size);
  }

  public Object getElement(int pos) {
    return elements[pos];
  }

  Object[] $snip$1(int size) {
    return new Object[size];
  }
}
```

Figure 4.5: Refined Java base class translation

```
Vector<String> vs = new Vector<String>(5);
String s = vs.getElement(0);

// Wrapper class and interface
interface $Vector$_String_$ {};
class $$Vector$_String_$ extends Vector implements $Vector$_String_$ {
  public $$Vector$_String_$(int size) {
    super(size);
  }

  Object[] $snip$1(int size) {
    return new String[size];
  }
}

// Translated example code
Vector vs = new $$Vector$_String_$(5);
String s = (String)vs.getElement(0);
```

Figure 4.6: Refined Java wrapper class and interface translation

Additional language features

Refined Java allows a programmer to declare that a parameterized class should use *covariant subtyping* for a given type parameter. When a class `C<T>` uses covariant subtyping, the instantiated type `C<String>` would be a subtype of the instantiated type `C<Object>`. Normally, instantiated types are *nonvariant*—different instantiations of the same parameterized class will never be subtypes.

Covariant subtyping is allowed only if the type variable T is not used as a parameter type to any methods and there are no public instance variables of type T. This restriction is necessary for covariant subtyping to be statically type-safe. However, there are few useful classes that will ever meet this restriction.

4.2.3 Delayed heterogeneous translation

[AFM97] is implemented through a heterogeneous translation to standard JVM bytecodes. The translation, though, is performed not by the compiler, but rather the *class-loader*—the component of the JVM that loads and interprets `.class` files. Whenever a new instantiation is used, the class-loader constructs the needed translated class. The `.class` file format is modified to provide the information needed for the class-loader to perform the translation.

The late translation approach has some significant benefits. Because a heterogeneous translation is performed, no run-time casts are needed, improving performance. Unlike a compiler-based heterogeneous translation, separate compilation is possible, as the compiler only generates one "object" file for a parameterized class; having only one file as the result of compilation also reduces network transport time.

This approach still carries major costs, though. Heterogeneous translation results in a larger memory footprint than homogeneous: the methods in the parameterized class are duplicated for every type used as an actual parameter. Because the class-loader's translation is dependent on extra information in the `.class` file, existing Java library classes cannot be used in a parametric fashion. Finally, while this approach does not alter the JVM's verifier or instruction set (reducing security concerns), it still requires changes to the Java runtime, and thus is not compatible with existing systems.

4.2.4 Modified JVM

The where clauses of [MBL97] can also be implemented using the translation methods presented in the previous sections. A homogeneous translation, however, is very complex, because of the use of where clauses to bound parameterization. The main problem is that the methods of a where clause do not "belong" to any class or interface, and thus cannot be invoked directly in the body of the translated class. Invoking a where-routine (*i.e.*, a method referred to in a where clause) thus requires numerous runtime casts, which imposes a substantial performance cost.

To avoid the performance costs of these casts, the JVM and `.class` file format were modified to support where clauses directly. The authors of [MBL97] report a performance improvement of up to 17% with the modified virtual machine; [Tho97], however, reported a smaller increase. Also, as changes were made to the `.class` file verifier, there might be security problems in the modified virtual machine. A detailed examination would be necessary to determine the soundness of both the proposed changes and the specific implementation of them. Finally, this approach does not allow for the leveraging of existing Java class libraries.

4.3 Brief evaluation of the proposals

This section gives a brief evaluation of the above proposals. Chapter 7 provides a much more detailed comparison and evaluation of the different language designs; Chapter 8 does the same for the different implementations.

4.3.1 Type system changes

The above proposals all succeed at improving Java's flexibility. However, we believe that they do not go far enough; F-bounded parametric polymorphism is still too restrictive, while virtual types do not provide the desired level of static type safety.

Consider, for example, the node classes for singly- or doubly-linked lists. One would like to make `DoubleNode` a subclass of `SingleNode` to not repeat the code common between the two classes. However, as Figure 4.7 shows, this is not possible in Java, even after adding parametric polymorphism. The main problem is that one is not able to change the types of fields, or the signatures of methods. In this example, `setNext()` must be completely rewritten in `DoubleNode`, instead of calling the version in `SingleNode` because a different field must be used to hold the next node. Similarly, `DoubleNode` must redefine `setNext(SingleNode<T> newNext)` to raise an error. (There is a way to provide similar functionality using F-bounded parametric polymorphism; see Section 7.2 for the details.)

Virtual types would be able to support having `DoubleNode` as a subclass of `SingleNode`. The resulting classes, though, may be used in a manner that is not statically type-safe.

4.3.2 Proposed implementations

Any implementation of an extended version of Java must make a compromise between performance and backwards compatibility. From a performance perspective, the best method of implementation is to modify the Java Virtual Machine to directly support the new features. Unfortunately, to do so would prevent that extended version from running on the millions of existing Java runtimes. In the long run, modifying the JVM is the best approach; however, such a modification will only gain acceptance if it is part of a revised Java standard.

```
class SingleNode<T> {
  SingleNode<T> next;
  T element;

  void setNext(SingleNode<T> newNext) {
    next = newNext;
  }
  ...
}

class DoubleNode<T> extends SingleNode<T> {
  DoubleNode<T> dNext;  // Can't change the type of next
  DoubleNode<T> previous;

  void setNext(DoubleNode<T> newNext) {
    dNext = newNext;     // Can't use super.setNext(newNext)
    if (newNext != null) {
      newNext.setPrevious(this);
    }
  }

  void setNext(SingleNode<T> newNext) {
    throw new TypeError();
  }
  ...
}
```

Figure 4.7: Inflexibility in extended Java

Because of this, we believe that the best approach for implementing an extension to Java is to translate extended programs to standard Java, allowing those programs to run on existing JVM implementations. The proposed translation schemes, though, have flaws that affect the usability of the language. The homogenous translations of Pizza and Generic Java do not properly support the use of parametric types in certain contexts, such as array creation or `instanceof` expressions. The mixed translation scheme of Refined Java alters the type hierarchy through the addition of wrapper classes and interfaces, complicating runtime examinations of the hierarchy, such as through the `isInstance` method of `Class` (see Section 4.2.2).[5]

[5]The method `boolean isInstance(Object o)` returns true if the actual parameter `o` is an `instanceof` of the type of the receiver. For the `Class` object that represents the type `Object`, `isInstance` always returns true, as all types in Java extend `Object`.

Chapter 5

`Rupiah`: Language Design

In this chapter, we present extensions to Java's type system intended to improve the flexibility and expressiveness of the language. Section 5.1 introduces the `ThisType` construct, which makes possible greater code reuse than Java currently allows. Section 5.2 discusses the typing issues that `ThisType` raises. Section 5.3 examines the combination of `ThisType` with constrained parametric polymorphism, and Section 5.4 the specific changes to Java required by the addition of `ThisType`. Finally, Section 5.5 discusses `Rupiah`'s rules for casting, and subtyping among parametric polymorphic types.

These type concepts are based on ideas from the languages $\mathcal{LOOM}$ [BFP97] and PolyTOIL [BSvG95].

Portions of this chapter originally appeared in [Bru97a].

5.1 `ThisType`

The keyword `this` is used in Java in method bodies of classes to represent the object executing the method; in essence, `this` allows an object to refer to itself. Suppose `this` occurs in the body of method `m` of class C, and that D is a class that extends C but does not override `m`. When `m` is defined as part of C, `this` implicitly has type (class) C, while in the same (inherited) method of D, `this` has type D.

Because subclasses in Java are also subtypes (*i.e.*, if D extends C, an expression of type D can be used in any context that expects an expression of type C[1]), this implicit change of type does not cause type errors. However, this implicit change does result in a loss of information. In particular, if a method returns `this`, or a clone of `this`, the type system will treat the value returned as being of type C, even though when executed by an object of type D it will return an object of type D. This loss of information is most evident in the

[1] Technically, this is true in Java only for assignments and parameters. In a conditional expression (*i.e.*, `x ? a : b`), for example, the return expressions must be of the same type.

`clone()` method of `Object`, which is declared to return an `Object` and therefore must be cast to the actual type of the object being cloned.

In Section 4.3.1, we saw that it is not possible in other extended versions of Java (with the exception of [Tho97]) to define a doubly-linked node by extending a singly-linked node, because of typing problems. Specifically, the type of the parameter to the method `setNext()` was incorrect in `DoubleNode`. Because the method was defined in `SingleNode`, `setNext()` takes a parameter of `SingleNode<T>`, even though in `DoubleNode` it should take a parameter of type `DoubleNode<T>`.

A solution to the above problems is to add a keyword `ThisType`[2] to represent the interface type (*i.e.*, the public methods and fields) of `this`. (See Section 5.4 for how the type of `ThisType` is determined in `Rupiah`.) `ThisType` allows a programmer to write more accurate types for the fields and methods of classes.

For a first simple example, suppose we are given

```
interface Cloneable {
  ThisType clone();
}
```

Any class implementing `Cloneable` will have a method `clone()` which will return an object of the same type as the receiver. If `x` is declared to have type T, where T extends `Cloneable`, then `x.clone()` is an expression of type T. Because `ThisType` stands for the type of `this`, when we send a message to an object, all occurrences of `ThisType` are replaced with the type of the receiver.

For our next example, we introduce revised classes for the linked-list node examples from Figure 4.7. The flexible meaning of `ThisType` ensures that the type of `next` and the parameter of `setNext()` *change* to reflect the different meanings of `this` when we move from the superclass to the subclass. Because of these automatic changes, the redefined `setNext()` in `ThisDoubleNode` is able to reuse the definition of `setNext()` from `ThisSingleNode`.

An astute reader will notice that using `ThisType` as a parameter results in a covariant type change in a method—a change that is not type safe, as was discussed in Section 2.2. Some additional constructs are required for `ThisType` to be safe; they are discussed in the next section.

It is worth noting that some other languages also support constructs like `ThisType`. The language Trellis/Owl includes a `MyType` construct to stand for the type of `me` (its equivalent of `this`), while Eiffel uses `like Current` for the type of `Current`. Eiffel, though, does not type-check this construct properly, allowing the user to write, compile, and run potentially dangerous code. Trellis/Owl, on the other hand, imposes restrictions that reduce the expressiveness of the construct.

[2]In $\mathcal{LOOM}$, `ThisType` is called *MyType*.

```
class ThisSingleNode<T> {
  ThisType next;

  void setNext(ThisType newNext) {
    next = newNext;
  }
  ...
}

class ThisDoubleNode<T> extends ThisSingleNode<T> {
  ThisType previous;

  void setNext(ThisType newNext) {
    super.setNext(newNext);
    if (newNext != null) {
      newNext.setPrevious(this);
    }
  }

  void setPrevious(ThisType newPrevious) {
    previous = newPrevious;
  }
  ...
}
```

Figure 5.1: Linked-list nodes using `ThisType`

Ada 95 also supports the equivalent of `ThisType`, though without adding a new construct. In Ada, if one defines a `Node` class in which a method or field mentions `Node`, then in subclasses all occurrences of `Node` are implicitly replaced by the subclass name. There are two drawbacks to this construct in Ada 95. First, the programmer will likely be surprised by this change, and in some cases might not want it to occur. Second, Ada does not type check this construct so that it is guaranteed to work properly in subclasses, and the compiler must therefore insert dynamic checks.

5.2 Typing `ThisType`

For `ThisType` to be type-safe, further changes are required to Java's type system. To properly call methods that have a parameter of type `ThisType`, we need to impose a restriction on the receiver of such a method call. Also, the relationship between subclasses and subtypes needs to be modified.

5.2.1 Exact types

Normally in an object-oriented language, if an expression has type `T`, at run time its value may be of any type which extends `T`. Sometimes, however, we want to have homogeneous data structures (*i.e.*, ones where all values included have exactly the same type). To allow the programmer to specify this we introduce "exact" types. If `T` is an interface, then declaring a variable to have type `@T` indicates that at run time, the public interface of the object held in the variable will consist only of the features in `T`. Similarly, a variable `v` of class type `@C` may only hold an object of run time type `C`; objects that are of a type that extends `C` may not be assigned to `v`.

Some care is needed in typing methods that return `@ThisType`. For example, consider:

```
interface Cloneable {
  @ThisType clone();
}
```

Here, any class implementing `Cloneable` will have a method `clone()` that will return an object of *exactly* the same type as the receiver. If `x` is declared to have type `@T`, where T extends `Cloneable`, then `x.clone()` is an expression of type `@T`.

If all we know is that `x` has type `T`—not an exact type—then the only thing we know about the value of `x` is that its interface extends the interface of `T`.[3] Since `clone()` promises to return something with exactly the same interface as the receiver, we know that `x.clone()` will return a value whose interface extends `T`. Thus, `x.clone()` is an expression with type `T`, not `@T`.

[3]In this example, something that extends `T` includes `T`.

Exact types are essential to ensure the safety of *binary methods*: methods that have one or more parameters of type `ThisType`. The handling of binary methods in `Rupiah` is discussed below.

5.2.2 The meaning of `extends`

One drawback of `ThisType` is that subclasses need no longer generate subtypes. For example, `ThisDoubleNode` (in Figure 5.1) is not a subtype of `ThisSingleNode`; one may not use a value of `ThisDoubleNode` in some contexts that expect a value of type `ThisSingleNode`. The following method illustrates the problem:

```
void breakit(ThisSingleNode<T> nd1, ThisSingleNode<T> nd2) {
  nd1.setNext(nd2);
}
```

If `aNd1` and `aNd2` are both nodes of the same type, then the call of `breakit(aNd1, aNd2)` would execute without a problem. Suppose, however, that `aDnd1` is a doubly-linked node while `aNd2` is singly-linked. The call `breakit(aDnd1, aNd2)` will not be safe because the message `setNext` will be sent to a doubly-linked node with a parameter that is a singly-linked node, while the typing rules require that the parameter type be the same as the receiver's. In particular, if this call were executed, a crash would occur when `aNd2` is sent the message `setPrevious`. `ThisDoubleNode`, then, cannot be a subtype of `ThisSingleNode`.

This failure of subtyping arises whenever the original type (interface) has a binary method; in those cases, the subclass does not generate a subtype. However, there is another relationship between interfaces, called "matching," that allows binary methods to be handled in a safe and flexible manner.

Matching, as defined in the language $\mathcal{LOOM}$ [BFP97] corresponds exactly to extension of interfaces when we have type `ThisType`. As a result all uses of `extends` between interfaces will now correspond to matching. (Recall, however, that in the special case that the type being extended has no binary methods, then *extends* also corresponds to subtyping.)

Subtyping was useful because if `S` is a subtype of `T`, then a value of type `S` could be used in any context that expected a value of type `T`. As we have seen, matching no longer corresponds to subtyping. Of what use is it to know that one type extends another? The key is that if `S` extends `T`, for `T` a known type, then this provides us with information on what messages can be sent to an object of type `S`.

For example suppose that interface `FNodeIfc` extends `ThisNodeIfc`, given in Figure 5.2. If `fnd` is a value of type `@FNodeIfc` then, because `FNodeIfc` extends `ThisNodeIfc`, `fnd.getNext()` is defined and will return a value of type `@FNodeIfc`. This follows because the `extends` relation between interfaces implies any value of type `FNodeIfc` must have a parameterless method `getNext` that returns a value of type `@ThisType`. Similarly we can

```
interface ThisNodeIfc<T> {
    T getVal();
    @ThisType getNext();
    void setVal(T newVal);
    void setNext(@ThisType newNext);
}

class ThisNode<T> implements @ThisNodeIfc<T> {
    protected T val;
    protected @ThisType next;
    public ThisNode<T>(T value, @ThisType newNext) {...}
    public T getVal() { return val; }
    public @ThisType getNext() { return next; }
    public void setVal(T newVal) { val = newVal; }
    public void setNext(@ThisType newNext) { next = newNext; }
}
```

Figure 5.2: A node using `ThisType` and exact types

determine that `fnd.setNext(newFnd)` is defined and is well-typed as long as the value of `newFnd` has type `@FNodeIfc`.

What happens to a message send involving `ThisType` if we don't know the exact type of the receiver? That is, if `gnd` has type `ThisNodeIfc`, then the value of `gnd` at run-time may have any type which extends `ThisNodeIfc`. As we saw earlier, in the case of a method like `getNext()` which has return type `@ThisType`, there is no problem. At runtime the variable `gnd` will have a value of some type matching `ThisNodeIfc`, call it `GNodeIfc`. Because `getNext` returns a value of type `@ThisType`, `nd.getNext()` will return a value of type `GNodeIfc`, which, by assumption matches `ThisNodeIfc`. Thus we can deduce that `nd.getNext()` has type `ThisNodeIfc`, indicating that the value is of some type matching `ThisNodeIfc`.

However we have more serious problems if the method has a parameter with type involving `ThisType`. Look again at `gnd.setNext(newGnd)`. Because `setNext` takes a parameter of type `@ThisType`, the type of the value of `newGnd` must be the same as the type of `gnd`. Thus if `gnd` has a value of type `@NodeIfc`, then the value of `newGnd` must be of type `@NodeIfc`. On the other hand if the value of `gnd` actually has type `@DbleNodeIfc`, then the value of `newGnd` must be of type `@DbleNodeIfc`. Since we do not know statically the exact type of `gnd`, we cannot determine statically what type the parameter must have to guarantee that the call will be type safe. Therefore we must rule out sending "binary messages" (i.e., messages corresponding to binary methods) to objects whose exact type is unknown. In other words, binary messages can only be sent to expressions whose type is of the form `@T`.

Using this information we see that the definition of the method `breakit` at the beginning of this section does not type check since we do not know the exact type of the receiver, `nd1`, of the `setNext` message, yet this is a binary method. However if we rewrite the example with exact types there is no problem:

```
public void nobreakit1(@ThisNodeIfc nd1, @ThisNodeIfc nd2)
{
   nd1.setNext(nd2);
}
```

A more general rewriting gives:

```
public void nobreakit2<N extends ThisNodeIfc>(@N nd1, @N nd2)
{
   nd1.setNext(nd2);
}
```

In either case the method type checks without difficulty since we know the exact type of the receiver. In the definition of `nobreakit1` the specification of an exact type ensures that we can only invoke it with a parameter which is a singly-linked node. In the case of `nobreakit2`, the parameters can be of any type, `N`, extending `ThisNodeIfc`, but both parameters must be of exactly that type.[4]

Despite the restriction that the target of a binary message must have an exact type, we will see that in many (perhaps even most) circumstances, matching is more useful than subtyping. We illustrate this in the next section where we revisit bounded polymorphic (or generic) classes where we take advantage of `ThisType`.

5.3 `ThisType` and parametric polymorphism

This example uses the node interfaces and classes from the previous sections to construct a class which can generate either singly or doubly-linked lists holding values of type `T`.

```
public class List<T, N extends ThisNodeIfc<T>> {
   protected @N head;
   public List<T,N>() { head = null; }
   public void addToHead(@N newNode) {
      newNode.setNext(head);
      head = newNode; }
   public T deleteHead() {
```

[4]Methods with type parameters are not yet implemented.

```
      T temp = head.getVal();
      head = head.getNext();
      return temp; }
   ...
}
```

The expression `new List<Integer, ThisNodeIfc<Integer>>` creates an empty singly-linked list capable of holding `Integers`, while `new List<Float, ThisDblNodeIfc<Float>>` creates an empty doubly-linked list capable of holding `Floats`.

Note that lists formed by instantiating `List` with type `T` hold elements of any type extending `T`. That is, it generates heterogeneous lists. Adding a few "`@`"s to appropriate occurrences of `T` in the node classes and `List` would result in homogeneous lists.

When we define ordered structures we need to be able to compare elements. The following interface lists the necessary operations.

```
interface OrdEltType{
   boolean gt(@ThisType other);
   boolean eq(@ThisType other);
}
```

We can now form a (homogeneous) ordered data structure using any type `T` which matches `OrdEltType`, since if two elements, `o` and `p` have type `@T`, then `o.gt(p)` will be defined and type-correct.

The fact that `extends` now corresponds to matching was quite important for both of these examples, as there are no non-trivial subtypes of `OrdEltType` or `ThisNodeIfc<T>` because of the occurrences of `ThisType` as a parameter in at least one method of each. Thus matching is essential in describing the constraint on the type parameter.

Generic subclasses are also easy to write with this setup. For example, the following defines an ordered list class which keeps track of a special "current" element.

```
class CurList <T extends NewBound,N extends ThisNodeIfc<T>>
              extends List<T,N> {
    @N current;
    ...
}
```

Note that we can make the bound of `T` smaller (in the "extends" ordering) than the one for `List` as long as the new bound extends the bound in the superclass. We follow with a more detailed discussion of the implications of these rules.

5.3.1 A note on instantiations

For a parameterized class

```
class C<T extends X>
```

the bound X may be either a class type or an interface type. If X is a class type, then only class types may be used to instantiate C. An interface type could not be used, as an interface type cannot extend a class type.

If X is an interface type, it might seem that one could only use an interface type to instantiate X, as only interface types extend interface types. In Rupiah, though, one may use a class type to satisfy an interface type bound. The logic here is that if a class type T implements an interface X (or a subinterface of X), then the interface of T (*i.e.*, ThisType) must extend the interface X. Therefore, both class and interface types may be used to instantiate parameterized types that have an interface type as the bound. Since class types can be assigned to interface types, no type problems can result from using a class type as an actual type parameter.

5.4 ThisType in Rupiah

In this section, we describe the specifics of ThisType in Rupiah. We first give a summary of the ThisType type rules.

5.4.1 Summary of type rules

To explain the type rules for ThisType, we must first briefly introduce the concept of *exact interface implementation.* A class C exactly implements an interface I if and only if the set of public methods of C is equivalent to the set of methods of I, including inherited methods. This means that the public interface of C is identical to I. Exact interface implementation is denoted by writing @I in the implements clause of the class declaration. A more detailed discussion of exact interface implementation can be found in Section 5.4.2.

The following rules summarize the typing of ThisType for

```
class C implements @IC { ... }
```

The exact interface @IC might not be explicitly declared; we use it to stand for the actual interface of C. The following relationships describe the relationship of ThisType to C and IC. For the purpose of explaining the type rules, we introduce a variable ThisClass to stand for the class type of C.

```
ThisType extends IC;
ThisClass extends C;
ThisClass implements @ThisType;
```

The type of `this` is `@ThisClass`. It follows from the above rules that `this` also has type `@ThisType`, since `ThisClass` exactly implements `ThisType`. Unlike `ThisType`, `ThisClass` is not a formal language construct. We use it to show that the type of `this` extends `C`, and that the interface of `this` is always exactly `ThisType`.

By these rules, we may assign `this` to a variable of type `C`, as `this` has type `ThisClass`, which extends `C`. We may also assign `this` to a variable of type `@ThisType`, as `ThisClass` exactly implements `ThisType`.

We may assign a value of type `ThisType` to a variable of type `IC`, as `ThisType` extends `IC`. We may not, however, assign a value of type `ThisType` to a variable of `@IC`.

We may not assign a value of type `ThisType` to a variable of type `C`, as `ThisType` is an interface type, not a class type. We also may not assign a value of type `C` to a variable of type `ThisType`, as `ThisType` extends the interface of `C` (*i.e.*, `IC`).

Finally, the only values that we can assign to a variable of type `ThisType` are those that results from an expression of type `ThisType`. These values may come from a variable, `this`, a parameter, or other expression. Because the receiver of a binary method must be of an exact type, the meaning of `ThisType` is fixed for the method call. We can therefore assign a value of the same type as the receiver to a parameter of type `ThisType`, as we know the precise meaning of `ThisType`. For example, in

```
class C implements @IC { ... }
@C c = new C();
c.aBinaryMethod(...);
```

we know that in the call `c.aBinaryMethod()`, a formal parameter with type `ThisType` requires an actual parameter of type `IC`, as `IC` is the interface for `C`.

5.4.2 The type of `ThisType`

In Java, an object created from class `C` can have many different types: `C`, the superclasses of `C`, the interfaces that `C` implements, and their superinterfaces. An object of type `C` can be assigned to a variable of any of the above types. A variable of type `C` can hold objects of type `C`, or objects of a type that extends `C`. A variable of an interface type `I` can be assigned objects that have type `I`.

Occurrences of `ThisType` in an interface `I` stand for the interface type `I`. This makes perfect sense: `ThisType` represents the interface of a type, and is thus identical to an interface type.

Because an object of a class can have multiple types, the exact meaning of `ThisType` for classes requires some explanation. Take, for example, the class definitions given in Figure 5.3. What types of objects can we assign to `v`, and to what types of variables can `v` be assigned to?

```
class SC {...}
class C extends SC implements I, J, K {
  ThisType v;
}
```

Figure 5.3: Using `ThisType`

The rules of the previous section state that the only thing that may be assigned to a variable of `ThisType` is an expression of type `ThisType`. Thus, we can assign to `v` the value from another variable of type `ThisType` (including `this`), from a parameter of type `ThisType`, or from a method with the return type `ThisType`.

By the definition of `ThisType`, `v` must have an interface that matches the interface of `C`. Objects that are to be assigned to `v` must therefore also have an interface that matches `C`'s. In the Java type hierarchy, the only types guaranteed to match `C`'s interface are the class `C` and classes that extend `C`. Conceivably, there might be another class `X` that matches `C`'s interface but does not extend `C`; it would seem that an object of `X` should also be assignable to `v`.[5] In Java, though, such relationships must be explicitly declared, so an object of such a type `X` may not be assigned to `v`. Furthermore, there is no possible way to declare such a relationship, as the interface of `C` does not have a name.

However, though `ThisType` in `C` matches the interface of `C`, the meaning of `ThisType` changes in subclasses of `C`. Because of this, one may not assign an object of type `C` to `v`—while this assignment is safe in the body of `C`, it would not be safe in a class that extended `C`. Say, for example, that `C` has the method

```
public void Danger() {
  v = new C();
}
```

that is not redefined in a class `D` that extends `C`. Remember that in class `D`, `ThisType` represents the interface of `D`, not `C`. If we had a variable `d` of type `D`, the method call `d.Danger()` would fail, as the method would attempt to assign an object of type `C` to a variable that represents the interface of `D`. Thus, we may not assign an object of type `C` to `v`, as that assignment will not be safe in a subclass of `C`.

Since the interface of `C` implicitly extends the interface types `I`, `J`, and `K`, `v` can be assigned to variables of those types. We cannot, though, assign `v` to a variable of type `C`, as interface types may not be assigned to class types. A class type, as opposed to an interface type, may contain instance variable or non-public methods. While `ThisType` represents the interface type of `C`, it does not represent the class type.

[5]This is known as *structural* matching.

Exact interface implementation

For a class C, the interface of C, from the compiler's perspective, is a somewhat nebulous concept. Recall that Rupiah, like Java, uses declared type relationships; that is, a class must explicitly declare what interfaces it implements. However, Java does not have a way to declare the complete interface for a class—there is no way to give the exact interface of a class C a name.

Not being able to name the interface for a class causes the following problem. Take, for example, the following definitions:

```
interface NodeIfc {
  void addNode(@ThisType node);
}

class Node implements NodeIfc {
  protected @ThisType next;
  public void addNode(@ThisType node) {
    next = node;
  }
}

class OtherNode implements NodeIfc {
  protected @ThisType next;
  public void addNode(@ThisType node) {
    next = node;
  }
}
```

Note that both Node and OtherNode have an identical interface (*i.e.*, both classes have only the methods declared in NodeIfc), and thus ThisType should be identical between the two classes. However, since we do not have a way to give a name to the interface for a class, there is no way to declare the relationship between Node and OtherNode. Because of this, we would not be able to write code such as

```
@Node n = new Node();
n.addNode(new OtherNode());
```

since we can not declare that Node and OtherNode have the same interface.

To solve this problem, Rupiah adds *exact interface implementation.* A class C exactly implements an interface I if and only if the set of public instance methods in C is identical to the set of methods in I. By public instance methods, we mean methods that are public, but are not constructors or static. The syntax for exact implementation is

```
class C implements @I {...}
```

As C and I have the same interface, ThisType is identical between the two. Therefore, ThisType for C and I stands for I, the common interface.

The correct way to write the above example is

```
class Node implements @NodeIfc {...}
class OtherNode implements @NodeIfc {...}
```

For both classes ThisType represents NodeIfc, so one would be able to use an OtherNode as the parameter to the addNode method of Node.

It is important to note that the set of public methods for a class includes both the methods declared in that class, and the public methods declared in the classes it extends; the same holds true for interfaces. When checking if a class satisfies the rules for exact implementation, the Rupiah compiler examines the full method sets. The compiler assumes that an interface implicitly includes the methods of Object. This view is justified by the fact that one can call a method from Object, such as toString(), on a variable that is an interface type.

5.4.3 The type of this

The multiplicity of types for a class C also means that this within C can be viewed as having different types.

The following example shows the various the roles of this. In the call to methodOne, this has type ThisType; in the call to methodTwo, this has type C. This follows from the type rules of Section 5.4.1, which give this the type of both the class and its interface.

```
class C implements @I {
  void methodOne(ThisType foo) { ... }
  void methodTwo(C bar) { ... }
  void test() {
    methodOne(this);
    methodTwo(this);
    @ThisType a = this;
    @C b = this;          // Illegal
  }
}
```

Assignment to variables of exact types is more complicated. In the first assignment statement, this has the type @ThisType. This will always be true, even in subclasses, as the meaning of ThisType will change in a subclass. The second assignment statement, however, is not correct. Suppose class D extends C, but does not redefine test(). When test() is called for an object of type D, this will also be of type D—but one may not assign an object of type D to a variable with type @C. Therefore, this has type @ThisType, but not @C.

5.5 Casts and subtypes

5.5.1 Parametric polymorphism and subtyping

If a class D extends a class C, then an object of D can be assigned to a variable of C. What rules should be used if class D<T> extends C<T>?

If V extends U, one might desire to be able to assign an object of D<V> to a variable of C<U>. This assignment rule, known as *covariant* subtyping, is not generally safe, for the same reasons that Java's rules for array subtyping are dangerous. If C and D have the methods

```
T one();
void two(T t);
```

then it follows from the rules of Section 2.1.2 that T must be the same type in C and D for an object of D<T> to be assignable to a variable of C<T>. Since one returns an object of T, T is able to vary covariantly. Since T is used as a parameter to the method two, T is able to vary contravariantly. As the only type that varies both covariantly and contravariantly from T is T, an *invariant* subtyping rule must hold for instantiated types to be safe in all circumstances.

It is possible for a subclass of a parameterized class C<I, J> to have a different number of type variables. For example, one could have

```
class D<K, L, M> extend C<L, K> {...}
class E extends C<Object, String> {...}
```

The invariant rule here is that corresponding type parameters must be identical. An object of E, then, is assignable to a variable of C<Object, String>, but no other instantiations of C. Similarly, an object of D<String, Object, Foo> can be assigned to a variable of C<Object, String>.

5.5.2 Casting rules

Rupiah's additions to the Java type system greatly reduce the need for type casts. However, programmers might still need to use such casts, especially when interacting with legacy code. To support those interactions, Rupiah includes casting rules consistent with current Java practice. Casts that will never succeed at runtime are rejected by the compiler. Casts that may fail at runtime are allowed, and a runtime check is performed.

Exact types

Values that are of the non-exact class type `C` or of a non-exact supertype of `C` may be cast to the type `@C`. This cast will be checked at runtime to ensure that the value is actually of the type `@C`. Values of a type that extends `C` may not be cast to the type `@C`, as such a cast would never succeed. The same rule holds for casting a value of a non-exact interface type to the interface type `@I`: `I` must extend that interface type.

By these rules, we may cast a value of type `Object` to type `@C`, as this cast might (but by no means is guaranteed to) succeed at runtime. We may not, though, cast a value of type `C` to type `@Object`, as such a cast would always fail.

Values of a non-exact interface type may be cast to an exact class type if and only if the class implements the interface. The class need not exactly implement the interface. (This is the same rule Java uses for casting a value of an interface type to a final class type.)

Values of a non-exact class type may be cast to an exact interface type if and only if all of the public methods of the class are present in the interface. The class does not need to implement the interface, as a subclass might exactly implement the interface. If the class had a public method not present in the interface, though, it would not be possible for a subclass to exactly implement the interface, so the cast would always fail at runtime. If the class type is final, then the class must exactly implement the interface.

Values that are of an exact class type may not be cast to any other exact class type. A value of an exact class type may be cast to an exact interface type if and only if that class exactly implements the interface.

Values that are of an exact interface type may not be cast to any other exact interface type. A value of an exact interface type may be cast to an exact class type if and only if that class exactly implements the interface.

Values of an exact class type may not be cast to a non-exact subclass, as such a cast is guaranteed to fail. A value of an exact class type may be cast to a non-exact supertype, or to a non-exact interface type that the class implements.

The standard Java casting rules are used for casting a value of an exact interface type to a non-exact class or interface type.

Polymorphic types

An instantiated type $\texttt{C<p}_1\texttt{, ..., p}_n\texttt{>}$ may be cast to a different instantiated type $\texttt{D<q}_1\texttt{, ..., q}_m\texttt{>}$ if and only if the following conditions hold:

- `C` and `D` satisfy the normal Java casting rules, based on their positions within the type hierarchy.

- The actual type parameters to C and D are identical for corresponding formal parameters.[6]

The second condition follows from the invariant subtyping rule for instantiated types. If corresponding type parameters did not have the same actual values, then there is no possible way that C could be a subtype of D.

These rules allows one to make a narrowing cast from, say, a Vector<String> to a Stack<String> (assuming Stack<T> extends Vector<T>. One may not cast an object of Vector<Object> to a value of Stack<String>, as the actual type parameters are not identical.

ThisType

Within a non-parameterized class, ThisType is, for the purposes of casts, treated as if it were a normal interface type. We use the standard Java cast rules for interfaces (see §5.5 of the Java Language Specification [GJS96]). Casts to and from @ThisType follow the rules for casts to and from exact interface types given above.

In a parameterized class, the cast rules for ThisType need to take into account the type parameters. It follows from the rules of Section 5.4.1 that ThisType extends the interface IC<T> of a class C<T>. By the rules of Section 5.5.1, the implicit parameter to ThisType in this case must be identical to the parameter to IC<T>. Thus, within a parameterized class, a value that is to be cast to ThisType must have been instantiated with the *formal* type parameters of the class. For example, within

```
class C<T extends D> implements @IC<T>
```

a value that is to be cast to ThisType must be instantiated with T, the class's formal parameter. A value instantiated with any fixed type may not be cast to ThisType.

[6] By corresponding formal parameters, we mean parameters that refer to the same type. For example, if class D<K> extend C<String, K>, then the first parameter of D, K, corresponds to the second parameter of C. The correspondence of formal parameters can only be determined reliably by looking at the declarations of the classes and interfaces in question (*i.e.*, looking at the syntax of the extends and implements statements).

Chapter 6

`Rupiah`: Implementation

6.1 Basic strategy and goals

The set of goals presented in Chapter 1 are, unfortunately, contradictory—it is impossible to design an implementation that will satisfy all of the goals. Unless one modifies the Java Virtual Machine, for example, it is impossible to support using primitive types (*e.g.*, `int`) as actual type parameters without needlessly duplicating code, or incurring a major runtime cost. Because of this, our implementation of `Rupiah` was designed to satisfy a more limited set of goals:

- No changes to the Java Virtual Machine.
- Parameterized types should be equivalent to conventional types.
  - Support for the `instanceof` operator.
  - Checked casts.
- Proper support for Java arrays.
- Avoidance of name mangling.
- Interoperability with existing Java classes and libraries.
- No needless duplication of code.
- Minimal performance costs.

Notable deletions from the original goals include using primitive types as parameters and security equivalent to Java's (though we do support a somewhat looser security model).

The implementation strategy is to translate `Rupiah` source code into standard JVM bytecodes. Because the JVM design is tightly coupled to that of the Java language, the majority of the translation mechanisms can be described by showing the resulting Java program, instead of a bytecode listing. Our implementation consists of a modified version of Sun Microsystem's `javac` compiler. The actual translations are performed at the internal program representation level.

6.2 Parametric polymorphism

Parametric polymorphism is supported through homogeneous translation, similar to the methods of Pizza and Generic Java. Each parameterized class results in a single `.class` file. To properly support certain language features, extra fields and methods are added to parameterized classes. An example of an ordered linked-list node and the resulting translation is given in Figures 6.1 and 6.2. (We do not use `ThisType` here. The implementation of `ThisType` will be discussed in Section 6.3.)

The first step in the translation is to replace type variables with the type of their bound. In the example, the variable `T` is replaced with `Object` in `Comparable` (as an unbounded type parameter has `Object` as the implicit bound) and `Comparable` in `OrderedNode`. A programmer simulating parametric polymorphism using the method described in Figure 2.4 would write code identical to this translation.

Next, a new method is added to support the `instanceof` operator. The method has name `$instanceOf$x`, where x is the name of the class or interface, and has the same number of parameters as the class. Each parameter is of type `EDU.williams.rupiah.PolyClass`, described below. The use and definition of these methods are described in Section 6.2.4.

The `$instanceOf$` methods need to encode the name of the class or interface to properly support inheritance and interface implementation. For example, if we had

```
class D<X, Y> extends C<X> implements I<Y> { ... }
```

an object of type `D<Object, String>` is an `instanceof` `D<Object, String>`, `C<Object>`, and `I<String>`. Class `D` therefore needs to have the following methods:

```
$instanceOf$D(PolyClass X$$type, PolyClass Y$$Type) { ... }
$instanceOf$C(PolyClass X$$type) { ... }
$instanceOf$I(PolyClass Y$$type) { ... }
```

As the method `$instanceOf$C` is already defined in class `C`, class `D` only needs to define the other two methods.

In classes, a private instance variable of type `PolyClass` is added for each type parameter to store its runtime type. This information is needed to support array creation, casting, `instanceof`, and per-instantiation static variables. In the example, the translated version of `OrderedList` contains the additional instance variable `private PolyClass T$$class`. The `PolyClass` objects must be declared private because subclasses do not necessarily have

```
interface Comparable<T> {
  boolean lessThan(T other);
  boolean greaterThan(T other);
  boolean equalTo(T other);
}

class OrderedNode<T extends Comparable<T>> {
  protected T data;
  protected OrderedNode<T> next;

  public OrderedNode<T>(T data, OrderedNode<T> next) {
    this.data = data;
    this.next = next;
  }

  public OrderedNode<T>(T data) {
    this(data, null);
  }

  public T getData() {
    return data;
  }

  public void setNext(OrderedNode<T> newNext) {
    next = newNext;
  }

  public OrderedNode<T> getNext() {
    return next;
  }
  ...
}
```

Figure 6.1: An example `Rupiah` class

```
import EDU.williams.rupiah.PolyClass;

interface Comparable {
  boolean lessThan(Object other);
  boolean greaterThan(Object other);
  boolean equalTo(Object other);
  boolean $instanceOf$Comparable(PolyClass T$$type);
}

class OrderedNode {
  protected Comparable data;
  protected OrderedNode next;
  private PolyClass T$$class;

  public OrderedNode(Comparable data, OrderedNode next,
                     PolyClass T$$$class)
  {
    T$$class = T$$$class;
    this.data = data;
    this.next = next;
  }

  public OrderedNode(Comparable data, PolyClass T$$$class) {
    this(data, null, T$$$class);
  }

  public Comparable getData() {
    return data;
  }

  public void setNext(OrderedNode newNext) {
    next = newNext;
  }

  public OrderedNode getNext() {
    return next;
  }

  public boolean $instanceof$OrderedNode(PolyClass T$$type) { ... }
  ...
}
```

Figure 6.2: The translated version of Figure 6.1

the same type parameters. Furthermore, the `PolyClass` objects are used to enforce runtime security, and therefore must not be accessible from outside of the class. As interfaces may not have instance variables, these PolyClass fields are not added to parameterized interfaces. (The `$instanceOf$` methods are added to interfaces.)

Finally, the constructors for parameterized classes are modified to properly initialize the `PolyClass` fields. For each type parameter `T`, a parameter `T$$$class` is added to every constructor. Within the body of the constructor, assignment statements are added to perform the needed initialization. In the above example, each constructor in `OrderedNode` gains a `PolyClass T$$$class` parameter.

For constructors that use a `this()` call to another constructor, no assignment is performed. Instead, the extra parameters are added to the `this()` call. In the above example, `this(data, null)` is translated to `this(data, null, T$$$class)`.

If a class extends a parametric class, then similar additions are made to the `super()` call that must be at the start of every constructor (except for those that refer to another constructor through a `this()` call). For example, if

```
class C<X, Y> extends D<X>
```

then the parameter `X$$$class` will be added to each `super()` constructor call, including the implicit call to the no-argument constructor if nothing is specified.

Finally, when a Java class does not define any constructors, a no-argument default constructor is added by the compiler. For parameterized classes, this synthesized constructor will contain the type parameters and the needed initialization assignment statements.

6.2.1 Storing type information

Runtime type information is stored in a `PolyClass` object, shown in Figure 6.3.[1] A `PolyClass` object stores type information in two separate components. First, it stores the `Class` object for the type. This variable refers to the internal representation of that type within the JVM.

Second, the `PolyClass` object contains an array of `PolyClass` to store any actual type parameters. This allows for instantiated types (*i.e.*, those that are parameterized) to serve as type parameters.

Ideally, identical instantiated types would be represented by a single instance of a `PolyClass` object, in the same way that two objects of the same type in Java are represented by the same `Class` object. This would make checking the equality of two `PolyClass` object references much faster, as we would only need to see if the two references were identical (*i.e.*, referred to the same object in memory). As the current implementation does not reuse `PolyClass` objects, a full structural equality check must be performed.

[1] This listing excludes one static method used to perform certain runtime checks. The method and its use is described in Section 6.2.4.

```
package EDU.williams.rupiah;
final public class PolyClass
{
    private Class base;            // Class object for the base class
    private PolyClass[] params;    // Our actual parameters

    public PolyClass(Class base, PolyClass[] params) {
        this.base = base;
        if (params != null) {
            this.params = new PolyClass[params.length];
            System.arraycopy(params, 0, this.params, 0, params.length);
        } else {
            this.params = new PolyClass[0];
        }
    }

    public PolyClass(Class base) {
        this(base, null);
    }

    public Class getBase() {
        return base;
    }

    public PolyClass[] getParams() {
        return params;
    }

    public boolean equals(Object o) {
        if (o instanceof PolyClass) {
            PolyClass p = (PolyClass)o;
            if (!base.equals(p.base)) return false;
            if (params.length != p.params.length) return false;
            for (int i = 0; i < params.length; i++) {
                if (!params[i].equals(p.params[i])) return false;
            }
            return true;
        }
        return false;
    }
}
```

Figure 6.3: The PolyClass object

Constructing PolyClass objects and checking for equality is linear with the total number of type parameters the object represents. An improved implementation that reuses the objects, instead of creating new ones, would greatly reduce the performance costs for object allocation (construction) and instanceof tests.

6.2.2 Using parametric classes

While the majority of translations need to be performed on classes and interfaces that have type parameters, some translation is also necessary for classes that use other parametric classes or interfaces. These translations consist of constructing the necessary PolyClass objects for constructor calls, and inserting casts for calls to methods that return an object that is of a variable type. In general, a cast needs to be inserted whenever the translated type is a superclass of the type used as an actual parameter.

The translation is slightly different when a type variable is used as an actual type parameter for an instantiation. In this case, instead of constructing a new PolyClass object, the appropriate $$class variable is used as the parameter to the constructor.

An example of these translations is given in Figure 6.4. The items of interest are the construction of a new PolyClass object for the constructor of Vector<String>, the use of the variable T$$class as an argument to the constructor of Vector<T>, and the casting of the return value of vs.getElementAt to String.

6.2.3 Arrays

As arrays in Java carry type information with them, one must be careful when allocating arrays of a variable type. If the array is not allocated with the correct runtime type, problems can result, as described in Section 4.2.1.

To create arrays with the correct runtime type, Rupiah makes use of the PolyClass object and the newInstance method of class java.lang.reflect.Array. The translations for array allocation are shown in Figure 6.5. The first parameter to newInstance, the Class object for the desired type, is taken from the corresponding PolyClass object. The second parameter, an array representing the dimensions of the array, is constructed from the dimensions given in the Rupiah version.

Java also allows partial array allocation. For example, in

```
String[][][] sa = new String[2][2][];
```

only the first two dimensions of the three-dimensional array are allocated. When translating an expression like this, a zero-length array is created for the unallocated dimensions; for the above example, we allocate a String[2][2][0]. This does not cause any problems,

```
class C<T>
{
    void aMethod(T t) {
        Vector<String> vs = new Vector<String>(5);
        Vector<T> vt = new Vector<T>(5);
        vs.insert(``Hello'');
        String s = vs.getElementAt(0);
    }
}

// Translated version
class C
{
    private PolyClass T$$class;

    void aMethod(Object t) {
        Vector vs = new Vector(5, new PolyClass(String.class));
        Vector vt = new Vector(5, T$$class);
        vs.insert(``Hello'');
        String s = (String)vs.getElementAt(0);
    }
}
```

Figure 6.4: Using parametric classes

```
// Rupiah version
class C<T extends I> {
    T[][] tArray;

    public void newArray(int a, int b) {
        tArray = new T[a][b];
        tArray = new T[5][];
    }
    ...
}

// Translated version
class C {
    I[][] tArray;
    private PolyClass T$$class;

    public void newArray(int a, int b) {
        tArray = (I[][])java.lang.reflect.Array.newInstance(
                     T$$class.getBase(),
                     new int[] {a, b});
        tArray = (I[][])java.lang.reflect.Array.newInstance(
                     T$$class.getBase(),
                     new int[] {5, 0});
    }
    ...
}
```

Figure 6.5: Rupiah array allocation

because Java's definite assignment rules require that this final dimension must be allocated before it is used.[2] In the above example, one needs to do something along the lines of

```
sa[0][0] = new String[1];
```

before assigning a value to `sa[0][0][0]`. This later (translated) allocation will replace the original zero-length allocation.

A note on accessibility

The actual parameter to a parametric class might not be *accessible* to that class. For example, if class `C<T>` is in package A and class `D` is private to package B (*i.e.*, it is not declared `public`), then `D` is not accessible from `C`. Within package B, though, one should be able to create an object of type `C<D>` without causing an access error.

Unlike the dynamic class allocation routine, `java.lang.class.newInstance`, the dynamic array allocation routine does not check to see if the type is accessible; that is, it does not throw an `IllegalAccessException`. The above translation, then, properly allocates arrays of types that are not accessible from the parametric class.

6.2.4 Casting and `instanceof`

In Java, one can determine the runtime type of an object through the `instanceof` operator. In `Rupiah`, the `instanceof` operator is extended to check whether an object is of a given instantiated type; we are able to differentiate between a `Vector<String>` and a `Vector<Object>`, for example.

The extended `instanceof` operator makes use of `PolyClass` objects and the synthesized `$instanceOf$` methods. The translation of the `instanceof` operator is shown in Figure 6.6. First, it checks to see that the object is of the proper base class, using the built-in `instanceof` operator. It then uses the `$instanceOf$` method to check if the object has the correct actual type parameters.

An `$instanceOf$` method has a `PolyClass` parameter for each type variable in the class. The method body simply checks to see if each `PolyClass` object passed in as a parameter is equal to the corresponding instance variable. If a class implements a parameterized interface `I<T>` with a fixed type `A` , the method checks to see if the base class in the `PolyClass` object is the same as the class object of `A`. Figure 6.7 shows examples of the different `$instanceOf$` methods a class may have.

The extended `instanceof` operator is also used to perform runtime checks for casts to instantiated types. When an object value `o` is being cast to type `I<T>`, `o` is first cast to `I`.

[2]However, if one did attempt to store a value in the "unallocated" array, an `IndexOutOfBoundsException` will be thrown instead of the expected `NullPointerException`. This might happen if the partially allocated multi-dimensional array is used as a parameter to a method.

```
// Rupiah versions
if (o instanceof Vector<String>) { ... }
if (o instanceof Vector<T>) { ... } // T is a type variable

// Translated version
if (o instanceof Vector
    && ((Vector)o).$instanceOf$Vector(new PolyClass(String.class)))
{ ... }

if (o instanceof Vector
    && ((Vector)o).$instanceOf$Vector(T$$class))
{ ... }
```

Figure 6.6: Translating `instanceof`

```
class D<X, Y> extends C<X> implements I<Integer> { ... }

// Translated version
class D extends C implements I {
  ...
  public boolean $instanceOf$D(PolyClass a, PolyClass b) {
    return X$$class.equals(a) && Y$$class.equals(b);
  }

  public boolean $instanceOf$C(PolyClass a) {
    return X$$class.equals(a);
  }

  public boolean $instanceOf$I(PolyClass a) {
    return a.getBase() == Integer.class;
  }
}
```

Figure 6.7: `$instanceOf$` methods

```
Vector<String> vs = (Vector<String>) o;
Vector<T> vs = (Vector<T>) o; // T is a type variable

// Translated version
if (!((Vector)o).$instanceOf$Vector(new PolyClass(String.class))) {
    throw new ClassCastException();
}
vs = (Vector) o;

if (!((Vector)o).$instanceOf$Vector(T$$class)) {
    throw new ClassCastException();
}
vs = (Vector) o;
```

Figure 6.8: Casts in Rupiah

This cast is checked by the JVM; a `ClassCastException` will be thrown if `o` is not of base type `I`. Next, the `$instanceOf$I` method is called on that cast value, with the appropriate parameter. If the method returns false, a `ClassCastException` is thrown; if it returns true, the cast has succeeeded. Figure 6.8 gives an approximate translation of the cast checks. The actual bytecode for the check does not translate to syntactically correct Java source, as the if statement is embedded within the assignment expression.

Side-effects

The above translations are incorrect if the object value being checked is the value returned from a method with a side-effect. The translations, as written, will call the method more than once, resulting in side-effect that the programmer did not intend to happen. For example, in the translation of

```
list.remove() instanceof Vector<String>
```

`list.remove()` will be called twice: once to check if the returned value is a `Vector`, and once to check that it has `String` as its type parameter.

This extra method call can be avoided through a bytecode translation that duplicates the needed value on the operand stack. This translation, though, is difficult to implement correctly because of restrictions placed on stack manipulation by the JVM's bytecode verifier.[3] In the absence of the correct translation, a simple workaround is to store the result

[3]Specifically, the requirement that the operand stack must be identical for all possible code paths to a given point.

of the method call in a temporary variable:

```
Object o = list.remove();
o instanceof Vector<String>;
```

(The internal structure of `javac` prevents us from automatically introducing the temporary variable.)

6.2.5 Per-instantiation `static` variables

A `static` variable is one that is shared across all instances of a class. Within a parameterized class, a static variable may be handled in two ways: each distinct instantiation can have its own copy of the variable, or a single copy can be shared among all instantiations. As distinct instantiations produce incompatible types (*i.e.*, an object of one instantiation cannot be assigned to a variable of another), the former behavior is preferable. The translations given above, though, result in the latter.

A possible translation to support the desired behavior is to replace a static variable of type `T` with a hashtable, and adding setter and getter methods to control access to the variable. The hashtable would be private to the class, and the methods would have the same access control as the variable. The setter and getter methods would take as parameters `PolyClass` objects representing the instantiation being used. These parameters would be hashed to access the correct value for that instantiation.

Figure 6.9 gives a tentative example of the translations. The current implementation does not make these translations; static variables are shared among all instantiations.

6.3 `ThisType`

The translations for `ThisType` are very similar to those for parametric polymorphism. The translated type for `ThisType` is the type that `ThisType` stands for, as given in Section 5.4.2. For an interface `I`, `ThisType` translates to `I`. For a class `C` that exactly implements `I`, `ThisType` also translates to `I`.

If a class `C` uses `ThisType` (*i.e.*, `C` has a variable or method parameter of type `ThisType`) but does not exactly implement an interface, the compiler will synthesize an exact interface for `C`. This synthesized interface will have the name `$$CIfc`, and will extend any interfaces that `C` is declared to implement. Also, if the superclass of `C` exactly implements an interface, then `$$CIfc` will also extend that interface.[4] Public methods in `C` that are not part of the interfaces `$$CIfc` extends will be added to `$$CIfc`; this means that `$$CIfc` will contain (through explicit declaration and extension) all of the public methods of `C`. For `C`, `ThisType` will translate to `$$CIfc`, as `C` now exactly implements `$$CIfc`.

[4] The exact interface of the superclass might also have been synthesized.

```
class C<T> {
  public static T foo;
  ...
}

C<T>.foo = C<T>.foo;

// Translated version
class C {
  private static Hashtable foo;
  public static void setFoo(Object value, PolyClass T) {
    foo.insert(T, value);
  }
  public static Object getFoo(PolyClass T) {
    return foo.get(T);
  }
  ...
}

C.setFoo(C.getFoo(new PolyClass(T.class)), new PolyClass(T.class));
```

Figure 6.9: Support for per-instantiation `static` variables

This translation presents a possible difficulty. If class `C` uses `ThisType`, but does not exactly implement an interface, and class `D` extends `C` and exactly implements the interface `ID`, then `ID` will not extend the interface `$$CIfc`. This will cause a problem if `D` does not redefine a binary method from `C`, and a user attempts to use, as a parameter to that method, a class that does not extend `C` but does implement `ID`.

Say, for example, `C` has the method

```
void foo(ThisType tt) { ... }
```

which translates to

```
void foo($$CIfc tt) { ... }
```

If we have a class `Q` that exactly implements `ID` but does not extend `C`, the following method call will fail, even though it is seemingly type-correct:

```
@D d = new D();
d.foo(new Q());
```

Since `Q` exactly implements `ID`, a value of type `Q` can be used as an actual parameter of type `ThisType` for the call `d.foo`. However, as `Q` does not implement `$$CIfc`, a value of type `Q` may not be used as an actual parameter to `foo`, as defined in `C`.

We note, though, that the same problem can result if a person uses only exact interface implementation, but does not correctly declare the inheritance relationships for the interfaces. One solution to this problem is to use structural matching. Structural matching, though, cannot be implemented efficiently for the standard Java Virtual Machine. If a programmer wishes to use multiple classes with the same exact interface, he or she must be careful to ensure that the exact interfaces declare the correct inheritance relations.

6.3.1 Runtime support for `ThisType`

To classes that use `ThisType`, the field

```
protected EDU.williams.rupiah.PolyClass $$ThisType
```

is added to store the proper runtime type for `ThisType`.[5] (Recall that the type of `ThisType` implicitly changes in subclasses, even for methods that are not redefined.) The field is set to the proper type within an object's constructor, as shown in Figure 6.10.

The `$$ThisType` field is used to properly create arrays (in the same manner as in Section 6.2.3) and to supplement the Java security model (see Section 6.6).

The `$$ThisType` field is also used to check casts to `ThisType` and `instanceof` expressions with `ThisType` on the right side. For example,

[5] As this is a protected field, it is inherited by subclasses. The field will only be added once; if one of the superclasses of a class uses `ThisType`, then this field will already exist.

```
class C {
  ThisType ttC;
  public C() { ... }
  ...
}

class D extends C implements Q {
  ThisType ttD;
  public D() { ... }
  ...
}

// Translated version
interface $$CIfc { ... }
class C implements $$CIfc {
  $$CIfc ttC;
  protected EDU.williams.rupiah.PolyClass $$ThisType;

  public C() {
    $$ThisType = new PolyClass($$CIfc.class);
    ...
  }
  ...
}

interface $$DIfc extends $$CIfc, Q { ... }
class D extends C implements $$DIfc {
  $$DIfc ttD;

  public D() {
    super();    // Implicit superclass constructor call
    $$ThisType = new PolyClass($$DIfc.class);
    ...
  }
  ...
}
```

Figure 6.10: `ThisType` translations

```
if (x instanceof ThisType) { ... }
```

translates to

```
if ($$ThisType.getBase().isInstance(x)) { ... }
```

Within a parameterized class, these checks make use of the `$instanceOf$` methods described in Section 6.2.4. For example, in a class `C<T>` that exactly implements `IC<T>` the above if statement would translate to

```
if (x instanceof IC
    && ((IC)o).$instanceOf$IC($$ThisType.getParams()[0]) { ... }
```

For these checks, though, we use the actual type parameter for `ThisType` that is stored within the `PolyClass` object.

6.4 Exact types

The translated version of a type `@T` is simply `T`. Additional checks must be added to properly support casting operations, however. The specific mechanics of the checks depend on whether the source and target of the cast are class or interface types.

When the target of a cast (*i.e.*, the type a value is being cast to) is an exact class type `@T`, we need to check that the source of the cast (*i.e.*, the value that is being cast) has runtime type `@T`. To do this, we simply compare the value's `Class` object with the `Class` object of `T`, as shown in Figure 6.11.[6] If the source of the cast does not have runtime type `@T`, then its `Class` object must be different from `C`'s. By definition, `null` may be cast to any reference type.

Checking a cast to an exact interface type `@I` is more complex. If the source value was also of an interface type, then a similar comparison of `Class` objects would suffice. However, it is impossible to have a value of an interface type, as one cannot construct interfaces—a runtime value must always have a class type. This check must therefore determine if the value is of a class that exactly implements the interface `I`.

The implementation of this check requires a small addition to classes that exactly implement an interface. To these classes, the `Rupiah` compiler adds the field

```
static private final boolean $$ExImp = true;
```

to signify that the class exactly implements an interface.

The check, shown in Figure 6.12, consists of three parts. Firsts, since a class that exactly implements an interface must directly implement one and only one interface, we make sure

[6]As with Figure 6.8, the translations shown here are an approximation of the actual bytecode emitted by the compiler.

```
class C { ... };
Object nonEC = new C();
@C exC = (@C) nonEC;

// Translated version
if (nonEC != null && C.class != nonEC.getClass()) {
  throw new ClassCastException();
}
exC = (C) nonEC;
```

Figure 6.11: Casting to an exact class type

that the runtime value meets this condition. To do this, we get the array of interfaces that the class implements by using the `getInterfaces` method of `Class`, and check to see that the array has length one. If the array does not have length one, then the check fails and we throw an exception.

Next, we check that this lone interface is the interface that is the target of the cast. We do this by comparing the `Class` object from the interface array to the `Class` object of the target interface. If the two `Class` objects are not identical, the check fails and we throw an exception.

Finally, we need to check that the source object contains the `$$ExImp` field. To do this, we get an array of the object's declared fields (*i.e.*, instance and static variables that are declared in the object's class, not inherited from a superclass).[7] To search the array for the desired field, we call the static method

```
boolean PolyClass.hasExImp(Field[] fields);
```

The definition of this method is given in Figure 6.13. If the method returns true, then the source value exactly implements an interface, and the check succeeds. If the method returns false, the check fails and we throw an exception.

As with the checks for parameterized casts, these translations are not correct when the value being cast is the return value from a method with side-effects.

6.5 Method binding complications

Because of method overloading and method redefinition in subclasses, there may be multiple methods in an object that can respond to the same message (method call). *Method binding*

[7] `Class` also contains a method that searches for a field with a specific name. However, this method throws an exception to signify failure, so it cannot be used as part of a boolean expression.

```
interface I { ... }
class C implements @I { ... }
I nonEI = new C();
@I exI = (@I) nonEI;

// Translated version
if (nonEI != null
    && (nonEI.getClass().getInterfaces().length != 1
        || nonEI.getClass().getInterfaces()[0] != I.class
        || !PolyClass.hasExImp(nonEI.getClass().getDeclaredFields())))
   {
      throw new ClassCastException();
   }
exI = nonEI;
```

Figure 6.12: Casting to an exact interface type

```
package EDU.williams.rupiah;
import java.lang.reflect.Field;

final public class PolyClass
{
    ...
    static public boolean hasExImp(Field[] fields) {
        if (fields == null) return false;
        for (int i = 0; i < fields.length; i++) {
            if (fields[i].getName().equals("$$ExImp")) {
                return true;
            }
        }
        return false;
    }
}
```

Figure 6.13: PolyClass.hasExImp(Field[] fields)

```
class C {
  void aMethod(Object o) { ... }
  void aMethod(String s) { ... }
  ...
}

C c = new C();
String str = ``A string'';
Object obj = str;

c.aMethod(str);  // aMethod(String s) chosen
c.aMethod(obj);  // aMethod(Object o) chosen
```

Figure 6.14: Resolving overloaded methods during compilation

is the process through which an object determines which method to execute in response to a message. This process can be static—the proper method is chosen at compiler time—dynamic—the method is chosen at runtime—or a combination of the two.

Java combines static and dynamic method binding. During compilation, overloaded method calls are resolved based on the parameter types of the message being sent. For example, in Figure 6.14, class `C` has two methods named `aMethod`; one takes an `Object` as a parameter, while the other takes a `String`. For the message `aMethod(x)`, the method that takes a `String` as a parameter will be chosen if a static type of `x` is `String` (*i.e.*, `x` is declared to be a `String` or a type that extends `String`). If `x` is declared to be any other type, the method that takes an `Object` as a parameter will be chosen. In Figure 6.14, the compiler will choose `aMethod(Object o)` for the call `c.aMethod(o)` because `o` is declared to have type `Object`. That fact that `o` will have type `String` at runtime does not matter; `aMethod(Object o)` will execute in response to the message, not `aMethod(String s)`.

In the output `.class` file, the compiler encodes method calls recording the name and *signature* of the selected method. A method's signature is a description of its return and parameter types. For example, a method that returns a `String` and takes an `Object` and an `Integer` as parameters would have the signature that we will write as `String(Object, Integer)`.[8]

At runtime, the JVM uses the method name and signature to determine which method body should execute, based on the runtime type of the receiver of the message. In Figure 6.15, both `C` and `D` contain a definition for the method `aMessage(Object)`. For the first

[8] This is not the actual syntax used to encode method signatures in the JVM. Using the real syntax would needlessly complicate the discussion.

```
class C {
  void aMethod(Object o) { ... }
  ...
}

class D extends C {
  void aMethod(Object o) { ... }
  void aMethod(Integer i) { ... }
}

C c;
Integer i = new Integer(5);
Object obj = i;

c = new C();
c.aMethod(obj);  // aMethod w/ signature void(Object) in C
c = new D();
c.aMethod(obj);  // aMethod w/ signature void(Object) in D

c.aMethod(i);    // aMethod w/ signature void(Object) in D
D d = new D();
d.aMethod(obj);  // aMethod w/ signature void(Object) in D
d.aMethod(i);    // aMethod w/ signature void(Integer) in D
```

Figure 6.15: Method binding in Java

call, the JVM executes the definition in `C` because the receiver is of runtime type `C`. For the second call, the JVM execute the definition in `D` because the receiver is of runtime type `D`—the static type `C` does not affect the process of selecting a method definition based on name and signature.

The latter three method calls of Figure 6.15 show the complications that result from Java's mix of static and dynamic method binding. In the call `c.aMethod(i)`, the receiver has static type `C` and the parameter static type `Integer`. In `C`, the only method with name `aMethod` takes a single parameter of `Object`, so the compiler chooses a signature of `void(Object)`. At runtime, the method `void aMethod(Integer i)` will not be executed, as it does not have the chosen signature. Choosing method signatures at compile time does not allow a more specific method in a subclass to be executed. The method `void aMethod(Integer i)` will be executed only if the receiver has static type `D` and the parameter has static type `Integer`.

6.5.1 Method binding problems in Rupiah

The above method binding complications in Java cause problems in the Rupiah translations in two different situations: when the bound on a type variable changes in a subclass, and when a class implements a parameterized interface using a fixed type, such as Integer, as the actual parameter for the interface.

Figure 6.16 gives an example of this problem. For the method call nsi.push(new Integer(5)) the compiler chooses the signature void(Number), as nsi has static type NumberStack. However, IntegerStack does not have a method push with the signature void(Number).[9] Because the method signature is determined at compile time, the correct method void push(Integer aNumber) is not executed. The same problem results with the call nsi.pop(): IntegerStack does not have a method that matches the signature Number(void).

This problem also manifests itself when the bound on a type variable is changed in a subclass. For example, if we had

```
class List<T> {
  void insert(T item) { ... }
  ...
}

class OrderedList<T extends Comparable> extends List<T> {
  void insert(T item) { ... }
  ...
}
```

the following code

```
List<Integer> li = new OrderedList<Integer>;
li.insert(new Integer(5));
```

would result in the execution of the insert method of List, as the compiler would choose the signature void(Object). The translated insert method of OrderedList has signature void(Comparable).

6.5.2 Bridge methods

To solve these method binding problems, the Rupiah compiler must insert extra methods during the translation process. This technique was first used by Pizza [OW97], which terms

[9]Technically, this means that IntegerStack does not implement NumberStack in the translated forms. The solution to this problem is identical to the solution to the method binding problem.

```
interface NumberStack<N extends Number> {
  void push(N aNumber);
  N pop();
}

class IntegerStack implements NumberStack<Integer> {
  public void push(Integer aNumber) { ... }
  public Integer pop() { ... };
}

NumberStack<Integer> nsi = new IntegerStack();
nsi.push(new Integer(5));  // want void push(Integer aNumber)
Integer i = nsi.pop();     // want Integer pop()

// Translated versions
interface NumberStack {
  void push(Number aNumber);
  Number pop();
}

class IntegerStack implements NumberStack {
  public void push(Integer aNumber) { ... }
  public Integer pop() { ... };
}

NumberStack nsi = new IntegerStack();
nsi.push(new Integer(5));        // Signature void(Number)
Integer i = (Integer)nsi.pop(); // Signature Number(void)
```

Figure 6.16: Method binding problems in Rupiah

```
interface NumberStack {
  void push(Number aNumber);
  Number pop();
}

class IntegerStack implements NumberStack {
  public void push(Integer aNumber) { ... }
  public Integer pop() { ... };
  // Bridge methods
  public void push(Number aNumber) {
    push((Integer)aNumber);   // calls signature void(Integer)
  }

  public Number pop() {
    return pop();             // calls signature Integer(void)
  }
}
```

Figure 6.17: Adding bridge methods to Figure 6.16

the additional methods *bridge methods*. These methods ensure that the correct method body is executed when the static method signature differs from the dynamic type.[10]

Figure 6.17 shows the translations of Figure 6.16 with the needed bridge methods inserted. The first bridge method, `void push(Number aNumber)` has the needed signature `void(Number)`. This method forwards the parameter to the `push` method with signature `void(Integer)`, inserting the appropriate cast.

The second bridge method, `Number pop()`, has the needed signature `Number(void)`. The method calls the `pop` method with signature `Integer(void)`, and returns the resulting value. No cast is necessary here, as `Integer` is a subclass of `Number`.

Adding the bridge method for `pop` results in the condition that the translated source for `IntegerStack` would not be accepted by a Java compiler, as the class contains two methods with the same name that differ only be their return type. This form of overloading is not legal as a call to such an overloaded method would be ambiguous—the compiler would not be able to choose a method signature to resolve to overloading. The `Rupiah` compiler is able to resolve this ambiguity, as it knows which method is a bridge method that it created. Bridge methods are never chosen when resolving a method call. Having

[10] They also ensure that the translated form of a class that implements an interface with a fixed type actually implements the translated interface.

two methods with the same name that only differ by return type does not cause a problem for the Java Virtual Machine because method signatures include return types; there is no ambiguity at the bytecode level as to what method should be executed.

6.6 Ensuring safety

One of the design goals of Java was to allow a person to run untrusted code without fear of damage. To make this possible, Java has a three-pronged security model. First, the language design prohibits certain dangerous operations, such as manual pointer arithmetic. Because of this, a programmer is unable to access arbitrary areas of memory.

Second, untrusted code (*i.e.*, code coming from an unknown source) is executed within a *sandbox*. Within the sandbox, certain operations, such as accessing the local file system, are prohibited. The sandbox restrictions make it impossible for code run within it to damage the underlying system.

Finally, the JVM runs a *bytecode verifier* on untrusted `.class` files. The verifier ensures that the untrusted code conforms to the Java language specification. This prevents a buggy or malevolent compiler from performing an illegal assignment, for example. In essence, the bytecode verifier duplicates the type-checking phase of a Java compiler.

However, the JVM's bytecode verifier is not aware of `Rupiah`'s extended type system, and is therefore unable to ensure that the new constructs in `Rupiah`—`ThisType`, exact types, and parametric polymorphism—are used correctly. Figure 6.18, from [AFM97], gives an example of the dangers that may result from this lack of full verification. At the position marked `(*)` in the example, a person may manually write bytecodes or use a faulty compiler to insert an `UnencryptedChannel` into `list`.

To supplement the protections the JVM's bytecode verifier affords, the `Rupiah` compiler inserts runtime checks into method bodies. These checks make use of the `PolyClass` objects and `$$ThisType` variable to ensure that actual parameters to methods are of the correct type. We emphasize that these checks will never fail for `.class` files from the `Rupiah` compiler, as the static type-checker guarantees the correctness of the program. The checks are needed to ensure that `Rupiah` program semantics are not violated by `.class` files not generated by the `Rupiah` compiler.

Figure 6.19 shows the added checks for parameters that are not of exact types. These checks make use of the `isInstance` method of `Class`. If the object passed as a parameter is not of the proper type, `isInstance` will return false and an exception will be thrown.

A much lengthier check must be used when the parameter is of an exact type. The check, shown in Figure 6.20, consists of three parts. First, the appropriate runtime type object is examined to see if the actual parameter is of a class or interface type. If the parameter is of a class type, then a check similar to that of Figure 6.11 is used. If the parameter is of an interface type, then a check similar to that of Figure 6.12 is used.

```
class BroadcastList<C extends Channel> {
  C channels[];
  void add(C c) { ... }
  void broadcast(String s) {
    for (int i = 0; i < channels.length; i++) {
      channels[i].send(s);
    }
  }
  ...
}

class EncryptedChannel extends Channel;
class UnencryptedChannel extends Channel;

BroadcastList<EncryptedChannel> list = ...;
list.add(new EncryptedChannel());
(*)
list.broadcast(``My password is...'');
```

Figure 6.18: Security dangers

```
class C<T> implements @I {
  void a(T t) { ... }
  void b(ThisType tt) { ... }
}

// Translated version
class C implements I {
  protected java.lang.Class $$ThisType;
  private EDU.williams.rupiah.PolyClass T$$class;

  void a(Object t) {
    if (!T$$class.getBase().isInstance(t)) {
      throw new ClassCastException();
    }
    ...
  }

  void b(I tt) {
    if (!$$ThisType.isInstance(tt)) {
      throw new ClassCastException();
    }
    ...
  }
}
```

Figure 6.19: Parameter checks for non-exact types

```
class C<T> implements @I {
  void a(@T t) { ... }
  void b(@ThisType tt) { ... }
}

// Translated version
class C implements I {
  protected java.lang.Class $$ThisType;
  private EDU.williams.rupiah.PolyClass T$$class;

  void a(Object t) {
    if (T$$class.getBase().isInterface()) {
      if (t.getClass().getInterfaces().length != 1
          || t.getClass().getInterfaces()[0] != T$$class.getBase()
          || !PolyClass.hasExImp(t.getClass().getDeclaredMethods()))
      {
        throw new ClassCastException();
      }
    } else {
      if (T$$class.getBase() != t.getClass()) {
        throw new ClassCastException();
      }
    }
    ...
  }

  void b(I tt) {
    if (tt.getClass().getInterfaces().length != 1
        || tt.getClass().getInterfaces()[0] != $$ThisType.getBase()
        || !PolyClass.hasExImp(tt.getClass().getDeclaredFields()))
    {
        throw new ClassCastException();
    }
    ...
  }
}
```

Figure 6.20: Parameter checks for exact types

The check for exact interface types, however, is not perfect. Recall from Section 6.4 that this check depends on a field added by the compiler. It does not attempt to locate each method of the parameter value in the exact interface, as such a check would be quite time consuming. Furthermore, because of the presence of bridge methods, it cannot always be determined at runtime whether or not a given method belongs to an interface.

Since the check for exact interface types does not actually examine methods, it is possible for a person to construct a class that will pass this check, even though it does not exactly implement the interface. It is not possible, though, to write a more stringent test without imposing a significant performance cost.

Unlike the casting checks of Section 6.4, the runtime checks for method parameters will not cause any side-effect problems. As Java uses pass-by-value semantics, these checks will always be on a single object value, and thus would never call a method multiple times.

Safety and class extension

A truly devious program can still manage to avoid the parameter checks described above. As the checks are inserted within method bodies, one can redefine those methods within a subclass, leaving out the security checks.

A possible way to avoid such trickery is to declare all non-private methods `final`. These methods would perform the necessary check, and then call a private method that would contain the actual method body. For example, a method `int insert(@T a)` would be translated to:

```
final public int insert(T a) {
  // The check for a
  return $$insert(a);
}

private int $$insert(T a) {
  // The real code for insert
}
```

Needless to say, such a translation increases the size of the `.class` files and greatly increases the costs of method calls. Because of these costs, we do not implement this translation.

Furthermore, this form of a security exploit is of minimal concern. In general, untrusted code is executed within an isolated environment—it may not interact with the rest of the system. Because of this, problems with such code are limited to that isolated environment. Though a programmer might be able to violate `Rupiah`'s semantics, he or she is still unable to escape from the constraints of the Java sandbox, and thus is prevented from causing lasting harm.

If one truly desires full security for an extension to Java, then modifying the JVM is the only possible choice. To do so, though, limits the number of people who will be able to use the extension. We feel that the slight security tradeoff of this implementation is worth the larger target audience.

Chapter 7

Evaluation of Language Designs

There are two main methods for evaluating a language design. The first method is to examine the design from a technical perspective—that is, how well the language satisfies certain theoretical criteria. Common evaluation factors include static type safety and the ability to reuse code.

The second method of language design evaluation concerns itself with the usability of the language—how easy (or difficult) is it to program using a given language? Languages that provide theoretically useful features but are difficult to reason about, and therefore use properly, will not be successful.

With this in mind, we now turn to the task to evaluating `Rupiah` and the extended Java proposals of Section 4.1. Our theoretical concern is how well the extensions increase the flexibility of Java, countered by their impact on static type safety. The corresponding evaluative factor is how well programmers are able to make use of this increased flexibility—do these changes actually make programming easier?

7.1 `Rupiah`

The combination of `ThisType`, exact types, and bounded parametric polymorphism provides a programmer with powerful and statically type-safe tools for building flexible and reusable classes and interfaces. These constructs allow a programmer to build complex container classes and to maximize the possible reuse of code. Take, for example, the linked-list example programs in Appendix B, where an ordered list is parameterized by both the type of the list element and the type of node (*e.g.*, singly- or doubly-linked). This form of parameterization works only because of the implicit covariant type changes of `ThisType`.

`ThisType`, though, does not come without a cost: exact types. For `ThisType` to be statically type safe, we are forced to limit binary messages (*i.e.*, calls to methods that have a parameter of type `ThisType`) to receivers that are of an exact type. Without this restriction, `ThisType` would suffer from the method parameter covariance problems described in Section 2.1.2. Thus, to ensure `Rupiah`'s static type safety, we must enforce the distinction between subclasses and subtypes when `ThisType` is used, and bar binary method calls to objects that are not of an exact type.

The need for exact types increases the difficulty of programming in `Rupiah`. When designing a data structure, a programmer must be aware of the restrictions of the use of `ThisType` and use exact types when necessary. The most difficulty involves designing classes that are easy to reuse and extend. Using exact types when they are not necessary will needlessly limit the possibility of reuse.

However, as non-exact types are the default, a programmer can use the compiler's error messages as an aid in choosing what types need to be exact. The programmer can simply make only those variables exact for which the compiler has flagged an invalid binary method call. This method of design-by-error is not possible in $\mathcal{LOOM}$, where exact types are the default. In `Rupiah`, a programmer must explicitly restrict the flexibility of a variable; in $\mathcal{LOOM}$, on the other hand, a programmer must choose where flexibility is possible.

`Rupiah` programmers must also be aware of the importance of exact interface implementation in designing reusable classes. Our preferred programming style is to use only interfaces for types, and have all classes exactly implement the corresponding interface. Explicitly declaring exact interface implementation makes it clear what `ThisType` stands for. More important, though, is that using exact interface implementation allows for two unrelated classes (*i.e.*, classes whose nearest common ancestor is `Object`) to be used in the same context. For example, one can use either a `CartesianPoint` object or a `PolarPoint` object for something of type `PointIfc`. Relying on synthesized interfaces when `ThisType` is used does not allow for this flexibility.[1]

When using exact interface implementation, a programmer must ensure that the interfaces used properly declare their inheritance relations. For example, if

```
class C implements @IC { ... }
class D extends C implements @ID { ...}
```

then `ID` should extend `IC`; otherwise, the compiler will emit an error. This requirement is necessary because if this were not the case, `ThisType` in `D` would not extend `ThisType` in `C`, violating the type-checking assumptions. Meeting this requirement presents some difficulty when `D` extends a class that has a synthesized exact interface (*e.g.*, `$$CIfc`), as the programmer will need to know some details of the translation model.[2]

Designing classes for safe reuse is a difficult task, though, no matter what language is used. The restrictions the `Rupiah` compiler imposes are there to ensure type safety. Less

[1] Using structural matching would allow for this flexibility without requiring exact interface implementation. However, this is difficult to implement efficiently without modifying the JVM.

[2] The error message from the compiler will give the needed interface name in this case.

restrictive languages, such as Eiffel, might appear to be easier to use to build reusable classes. However, the programmer must be aware of the type dangers inherent in such languages, so building safe and reusable classes can be even more difficult, as there will not be any compile-time errors to aid in the task.

7.2 F-bounded parametric polymorphism

Adding F-bounded parametric polymorphism to Java goes a long way in solving the languages most pressing need, better support for container classes. F-bounded parametric polymorphism still imposes restrictions, though, as it does not provide any means to change the types of instance variables or methods in subclasses.

Without this ability, F-bounded parametric polymorphism does not allow for easy and safe extension of classes that contain binary methods. Recall the linked-list node examples given in Figure 4.7. Because the programmer is not able to change the type of the variable `next` or the parameter for `setNext` in class `DoubleNode`, he or she is forced to duplicate the code for `setNext` from `SingleNode`. Furthermore, the programmer must also define a second version of `setNext` to check if a parameter of the wrong type is given, as there is no way for this error to be detected by the compiler.

While F-bounded parametric polymorphism does not make it possible for the class `DoubleNode` to extend the class `SingleNode`, it does allow for the creation of an ordered linked-list that is parameterized by the type of node, as shown in Figure 7.1. To accomplish this, the interface `Node<T, N>` takes as parameters the element type *and* the node type (*i.e.*, singly- or doubly-linked), where the node type must implement the `Node` interface. The class `SingleNode<T>` implements this interface. The interface for a doubly-linked list, `DNode<T, N>` extends `Node<T, N>`. Here, the node type must implement the interface `DNode`. The class `DoubleNode<T>` implements this interface.

The class defining the list takes as parameters the element type and the node type:

```
class OrdList<T implements Ord<T>, N implements Node<T, N>>
```

where the elements type `T` must implement the interface `Ord<T>`, and the node type `N` must implement the interface `Node<T, N>`. We can use a `SingleNode` as the node parameter to `OrdList`, as `SingleNode` correctly implements the interface `Node`. We can also use `DoubleNode` to instantiate `OrdList`, as `DoubleNode` correctly implements the interface `DNode`. Since `DNode` extends `Node`, `DoubleNode` also implements `Node`, and thus satisfies the bound of `OrdList`.

Because `DoubleNode` does not extend `SingleNode`, though, code must be duplicated. Also, the bounds for `Node`, `DNode`, and `OrdList` are complex. This functionality can be expressed with greater ease and clarity through the use of `ThisType`.

```
interface Ord<T> {
    boolean lessThan(T other);
    boolean equals(T other);
}

interface Node<T implements Ord<T>, N implements Node<T, N>> {
    void setNext(N newNext);
    N getNext();
    T getValue();
}

class SingleNode<T implements Ord<T>> implements Node<T, SingleNode<T>> {
    protected T value;
    protected SingleNode<T> next;

    public void setNext(SingleNode<T> newNext) { ... }
    public SingleNode<T> getNext() { ... }
    public T getValue(); { ... }
    ...
}

interface DNode<T implements Ord<T>, N implements DNode<T, N>>
               extends Node<T, N>
{
    void setPrevious(N newPrevious);
    N getPrevious();
}

class DoubleNode<T implements Ord<T>> implements DNode<T, DoubleNode<T>> {
    protected T value;
    protected DoubleNode<T> next;
    protected DoubleNode<T> previous;

    public void setNext(DoubleNode<T> newNext) { ... }
    public DoubleNode<T> getNext() { ... }
    public T getValue(); { ... }
    public void setPrevious(DoubleNode<T> newPrevious) { ... }
    public DoubleNode<T> getPrevious() { ... }
    ...
}

class OrdList<T implements Ord<T>, N implements Node<T, N>> { ... }
```

Figure 7.1: F-bounded parameterized linked-lists

Finally, F-bounded parametric polymorphism, by itself, does not allow for changes in the return types of methods.[3] When calling the `clone` method, for example, the programmer must cast the return value to the proper type, as clone is declared to return an `Object`. A `clone` method defined to return a value of type `ThisType` does not require such a cast.

7.3 Where clauses

Where clauses provide features similar to those of F-bounded parametric polymorphism, and thus present the same problems. Because the where clause proposal does not allow for types to change in subclasses, programmers are faced with the difficulties described above.

The difference between where clauses and F-bounded parametric polymorphism lies in how the compiler determines if a class meets the bound for a parameter. Bounds described through where clauses are structural—they do not depend on declared type hierarchies. The main benefit of structural matching is that a constraint can be written at a later time than a class that meets the constraint. For example, a class `ComplexNumber` might have `lessThan` and `equals` methods, but not declare itself to implement any interfaces. `ComplexNumber` would therefore not meet the bound `Comparable` (where `Comparable` is an interface that contains only the `lessThan` and `equals`), even though it possesses the relevant methods.

Structural matching presents two problems, though. First, there is the possibility of accidental conformance: a method with a given name and type might not have the desired semantics. An accidental semantic mismatch is much less likely to occur when a programmer implements a method listed in a constraint interface. Second, implementing structural matching through translation (as opposed to modifying the JVM) imposes a larger performance cost than translations that use interfaces as bounds.

Another justification given for where clauses is that using interfaces as constraints clutters the type hierarchy. Types meant as constraints, such as `Comparable`, are not likely to be used for any other purpose than as a constraint. Constraint interfaces greatly increase the number of subtype relationships that must be declared—according to [MBL97], the number of subtype relationships is roughly proportional to the number of parameterized types and to the total number of types in the system—possibly resulting in a decrease in performance.[4]

Unlike interfaces, where clauses can mention constructors. This makes it easier to construct an object of a parametric type with a parameterized class, as the type used as an actual parameter will be guaranteed to have the necessary constructor. Such a constraint, though, limits actual parameters to non-abstract classes (if the constructor is called within

[3] Generic Java, unlike Pizza, does allow for the change of method return types.

[4] [MBL97] makes this claim for "most implementations of object-oriented systems." We do not believe this to be the case for current implementations of Java, though we have not performed any experiments to determine the impact of declared subtype relationships to performance.

the body of the parametric class), and thus restricts the possible uses of the parametric class.

Where clauses also possess some general usability problems, especially for large systems. When the same constraint is used for multiple classes, the entire set of where clauses must be duplicated. For large constraints, this duplication is time consuming and possibly error prone. More important, though, is that changing the constraint requires one to duplicate the change for each individual class, instead of making a single change to the constraint interface.

Furthermore, a class `C`'s conformance to the constraints of another class `D<T>` is not checked until `C` is used to instantiate `D`. The compiler cannot determine whether or not `C` will satisfy a given bound when `C` is compiled, as the specifics of the bound might not be available (*e.g.*, if `C` and `D` were written in two different locations, and are being used together in a third). When interfaces are used as constraints, the compiler can determine that `C` satisfies a given bound when compiling `C`, as `C` must declare the interfaces that it implements. Using interfaces as constraints thus allows errors to be detected earlier.[5]

This benefit of early error detection is not lost even if interface constraints are combined with structural matching. With structural matching, a class `C` does not need to declare that it implements an interface `I` to satisfy the bounds given by `I`. If a programmer knows ahead of time, though, that `C` is meant to satisfy the bounds of `I`, he or she can declare `C` to implement `I` to gain the benefits of additional compiler-time error checking. It is not possible to gain this benefit when where clauses are used as bounds, as there is no way for a programmer to declare what bounds `C` is meant to satisfy.

Finally, interface constraints are more readable than those expressed through where clauses, especially for large constraints. For example, using `N extends NodeIfc<T>` as a constraint clearly states the intention of the bound: the actual parameter must be a node of `T`. The same constraint expressed through where clauses is not as clear in meaning:

```
where N {
  void setNext(N newNext);
  N getNext();
  void setValue(T value);
  T getValue();
}
```

Though one can fairly easily deduce that these methods represent a linked-list node, it requires more time and effort than interpreting an interface name. This problem increases in

[5]The programmer, of course, must declare `C` to implement the correct interface—it is possible that one would declare `C` to implement the interface `J` instead of the actual bound interface `I`, with the incorrect belief that `J` extends `I`. In this case, the error would not be detected until `C` was used to instantiate the bounded class.

severity for more complex constraints, especially when a single class has mutually recursive bounds. Compare, for example, the following Rupiah constraints from Appendix B.1:

```
class OrderedList<T extends OrdEltIfc, N extends ExactNodeIfc<T>>

class OrderedList[T, N]
where T {
  boolean gt(@ThisType other);
  boolean eq(@ThisType other);
} N {
  @T getValue();
  void setValue(@T newValue);
  @ThisType getNext();
  void setNext(@ThisType newNext);
}
```

Using interfaces to express constraints produces more concise and understandable code.

7.4 Virtual types

Unlike F-bounded parametric polymorphism and where clauses, virtual types provide more flexibility than the type system of Rupiah. Virtual types, though, can be used in ways that are not statically type safe. For container classes, they are harder to use and less readable than parametric polymorphism.

The main benefit of virtual classes is that they allow for the extension of mutually recursive systems of classes. Such systems are often found in standard design patterns [GRJV94]. However, the implementation [Tho97] described does not fully support this capability. Also, it is possible to support the extension of mutually recursive classes in a statically-typed language; see [Bru97b].

Because virtual types allow unrestricted covariant type changes in subclasses, they generate classes that can be used in a type-unsafe way, as shown in Figure 4.3. [Tho97] justifies the lack of static type safety with the claim that Java itself is not fully statically type safe. There are two possible places where a runtime type failure can occur in Java: failed casts and invalid array stores (see Figure 3.1). Both of these cases, though, are clearly visible in Java's syntax; a programmer is aware of the possible danger of the operation. Virtual types do not provide this visibility—a programmer may not be aware of the danger of a certain operation.

Virtual types also obscure the variable nature of container classes and make it more difficult to create containers that store objects of different type. For example, to create a Vector that holds Strings, a programmer would have to write the code in Figure 7.2.

```
class Vector {
  typedef ElemType as Object;
  ...
}

class StringVector extends Vector {
  typedef ElemType as String;
}

StringVector vs = new StringVector();
```

Figure 7.2: Virtual types and container classes

A programmer, though, might choose a different name for the resulting class, such as `VectorStr`, `StrVec`, or `Foo`. Because of this, it is possible that, in a large project, there might be multiple definitions of the same type of container.

Such multiple definition cannot occur using parametric polymorphism. The only way to declare a `Vector` of `String` is `Vector<String>`. Furthermore, this syntax makes explicit the underlying language mechanisms, and thus the possible dangers or performance costs. The fact that a class contains a virtual type is evident only in the class definition—a programmer might not be aware that virtual types were being used.

Finally, virtual types cannot capture desired subtype relationships among container classes. Take, for example, a class `Vector` that extends a class `Container`. To create a `Container` of `String`, one would have to extend `Container`, resulting in the class `StrContainer`. To create a `Vector` of `String` one would have to extends `Vector`, resulting in the class `StrVector`. An object of `StrVector`, though, may not be assigned to a variable of type `StrContainer`, as `StrVector` is not a subclass of `StrContainer`. The desired relationships are easily captured using parametric polymorphism:

```
class Vector<T> extends Container<T> { ... }
Container<String> cs = new Vector<String>; // legal
```

The mechanisms of parametric polymorphism are well known to programmers and explicitly visible within programs. Virtual types, on the other hand, are not well known. Their syntax also obscures their use within a program. Because of this, we believe that parametric polymorphism fits the design of Java more than virtual classes do, and thus are preferable as an extension to Java.

Chapter 8

Evaluation of Implementations

Ideally, the implementation of an extended version of Java would be backwards compatible with existing Java Virtual Machines, safe, efficient with respect to both time and space, and allow type parameterization with primitive types. Unfortunately, it is not possible for any implementation to meet all of these criteria.

In this chapter, then, we evaluate and compare the different proposed implementations. Because compatibility with existing Java installations is one of our major design goals, we place more emphasis on implementations that do not require changes to the JVM.

This chapter concludes with a proposal that combines some of the features from the various implementations. This combination should provide the best possible compromise among the above goals.

8.1 `Rupiah`

The current `Rupiah` implementation provides a wealth of features, including:

- Efficient use of space.
- Proper array creation for parametric types and `ThisType`.
- Support for checked parametric casts.
- Support for checked casts to parametric types.
- Support for a parametric `instanceof` operator.
- Reasonable runtime safety for parametric polymorphism, `ThisType`, and exact types.
- Minimal name mangling.

- Compatibility with existing JVM implementations.

It is not without its faults, though.

The current implementation imposes a non-trivial performance cost. There are three main areas where the costs are evident. First, if a method returns a value of a variable type (*e.g.*, a method in `C<T>` that returns a value of type `T`), then the compiler must insert a cast for that value to be used (see Figure 6.4). The exact cost of these casts depends on the JVM implementation. They should conceptually be of constant time, but we have not performed any experiments to assess their impact. There is no way to reduce these costs without modifying the JVM.

Second are the runtime checks for parameters of exact types, `ThisType`, and parametric type variables. These checks both reduce performance and increase the size of a `.class` file. The check for each parameter should take a constant amount of time, though this is also implementation dependent. Methods that take multiple parameters might therefore have a significant performance loss due to the runtime checks.

As the costs of these parameter checks can be quite large, we plan on adding an option to the compiler to prevent them from being added. This will allow installations that are only using trusted `Rupiah` programs to avoid these costs, similar to how a person may turn off the JVM's bytecode verifier for local programs. Achieving runtime safety without imposing significant performance costs will require modifying the JVM.

The final performance hit is for the construction of parametric objects. The main cost is due to the need to construct a `PolyClass` object. The cost for parametric type constructors depends on the number of type parameters and the types used as actual parameters. Using an instantiated type as a type parameter will result in a larger performance cost, as nested `PolyClass` objects will be constructed. The need to construct `PolyClass` objects also imposes a cost for parametric casts and `instanceof` checks.

The current implementation of the `PolyClass` objects is quite naive, in that it makes no attempt to reuse objects that have already been constructed. A more sophisticated implementation that only allocates a single `PolyClass` object for a given instantiated type (in the same way that the JVM only allocates a single `Class` object for a given type) will markedly improve performance and reduce memory usage.

The translations for per-instantiation static variables, when implemented, will also carry a fairly significant runtime cost, as a single variable access will translate into a number of method calls. We feel, though, that references to static variables for parametric classes will be rare, so that the cost of this translation will not have a major effect on actual programs.

Our implementation does not allow primitive types to be used as type parameters. A programmer must manually wrap primitive types in the appropriate object type (*e.g.*, `Integer` for `int`). There are two techniques to avoid the need for wrapping primitive types in objects: modify the JVM, or generate a new class for each possible primitive type instantiation. The second technique can result in a substantial increase in code size and memory usage, and makes separate compilation problematic.

8.2 Pure homogeneous translation

Pizza [OW97] and Generic Java [BOSW97] (Section 4.2.1) use a very simple translation model. Within a parametric class `C<T extends X>`, all occurrences of the type `T` are replaced with the type of the bound, `X`. (`Object` is used if no bound is present.) No runtime type information is stored. As with `Rupiah`, the compiler inserts casts when a method returns a value of a variable type. The only performance cost of this implementation strategy is that of the runtime casts.

However, the simplicity of these translations places substantial restrictions on the use of parametric types. Because no runtime type information is stored, there are a large number of desirable features that cannot be supported:

- Checked parametric casts.
- A parametric `instanceof` operator.
- Correct creation of arrays of parametric types.
- Per-instantiation static variables.
- Runtime safety.

The most dangerous of these problems is the lack of support for creating arrays of parametric types. Figure 8.1 provides an example (from [BOSW97]) of the dangers of this translation method. Within the body of the class `Vector<T>`, the array of type `T[]` is created with type `Object[]`. As Java arrays are tagged with their runtime type, this means that the attempt to cast that array to type `String[]` will fail, as the array is not of type `String[]`.

The problem here is not Java's covariant subtyping rule for arrays, as [BOSW97] claims. Instead, the problem is that the Pizza and Generic Java translations are not creating the arrays properly; the same problem would result even if Java had invariant arrays.

The lack of checked parametric casts will cause programs to fail at the wrong point in time. Instead of throwing an exception at the invalid cast, a Pizza or Generic Java program will throw a cast exception when the program attempts to remove an item from the container (*i.e.*, at a point where there is a compiler inserted cast).

One advantage that this implementation has over the current `Rupiah` implementation is its ability to interoperate with existing Java libraries and compilers. Standard Java programs can call Pizza classes without having any knowledge of the translation model—the programmer only needs to insert the appropriate casts. A Pizza compiler can also treat standard Java collection classes, such as `java.util.Vector`, as if they were parameterized,

```
class Vector<T> {
  private int n;
  private T[] ot;
  ...
  public T[] toArray() {
    T[] nt = new T[n];
    System.arraycopy(ot, 0, nt, 0, n);  // Copy ot into nt
    return nt;
  }
}

String[] as = new Vector<String>.toArray();

// Generic Java and Pizza translations
class Vector {
  private int n;
  private Object[] ot;
  ...
  public Object[] toArray() {
    Object[] nt = new Object[n];
    System.arraycopy(ot, 0, nt, 0, n);  // Copy ot into nt
    return nt;
  }
}

String[] as = (String[]) new Vector().toArray();  // Runtime error
```

Figure 8.1: Problems with arrays and Generic Java

inserting casts where needed. For example, if a variable `v` has type `Vector<String>`, the compiler would cast the return value of the method call `v.elementAt(0)` to a `String`.[1]

Some care is required for the latter use, though, as not all instances of `Object` in a container class refer to the type of the values being stored. For example, `Vector` has a method `clone()` that returns a value of type `Object`. This value, though, should be cast to a `Vector`, as it refers to the vector itself instead of an element of the vector. If the compiler were to cast the return value of `clone` to any type except `Vector`, a runtime error would result. We do not know if the current Pizza implementation makes this error; we simply wish to point out the possible danger.

These backwards-compatibility features can be added to the `Rupiah` implementation. Section 9.2 gives a brief description of how these translations would work, and of their limitations.

8.3 Mixed translations

The main goal of the Refined Java [CJ98] implementation is to correct the flaws present in the Generic Java scheme, without losing backwards compatibility with the existing Java virtual machine. While this scheme is quite different in design than our implementation, it provides similar features.

The main feature lacking in our implementation that the Refined Java implementation provides is support for the construction of objects of a parametric type within the body of a parameterized class (*e.g.*, constructing an object of type `T` within a `Vector<T>`). However, to construct an object of type `T`, `T` must be a non-abstract class type. Thus, if a Refined Java parametric class `C<T>` creates an object of type `T`, then only non-abstract classes may be used to instantiate `C`. In this case, Refined Java requires the programmer to write `class C<class T>` to indicate that only non-abstract classes may be used to instantiate `C`.[2]

It is possible in `Rupiah` to create objects within a parameterized class through a carefully designed constraint. If a programmer needs this capability, he or she can add a method

```
T createNewObject(...)
```

to the constraint interface. A class implementing the method would simply forward the parameters to the appropriate constructor, and return the reference to the newly created object. Explicit support for constructors within parametric class bodies is not needed. Furthermore, the programmer does not need to limit the types that may be used to instantiate such a class; the only requirement is that the type properly implements the constraint interface.

[1]Here `Vector` refers to the standard `java.util.Vector`, *not* a parametric version.

[2]Refined Java also allows `interface` or `abstract class` to preface a type variable. Using `interface` allows any type to be used to instantiate the class; `abstract class` prevents the use of interface types, but allows both abstract and non-abstract class types.

Refined Java also adds support for covariant subtyping for parametric classes (*i.e.*, `C<String>` would be a subtype of `C<Object>`). However, this form of subtyping can be used only if the class does not contain any methods that have as a parameter an object of a variable type (*i.e.*, in `C<T>`, no methods may have a parameter of type `T`). This is not a likely occurrence.

The Refined Java implementation imposes a minor restrictions on parametric classes. As constructors in wrapper classes need to call the appropriate constructor in the base class (through a `super()` call), parametric classes may not have private constructors. This restriction has little, if any, effect on the usability of the language.

In general, the Refined Java implementation depends on translations to static language features, while `Rupiah` makes use of Java's dynamic capabilities (*i.e.*, reflection). Performance for both implementation schemes should be comparable, depending on the strictness of runtime parameter checks and the specifics of the JVM implementation.

The translations required by the Refined Java implementation, however, are much more complex than those for `Rupiah`. The translation scheme involves multiple modifications to the source type hierarchy, due to the presence of wrapper classes and interfaces. These modifications might cause problems for programs that attempt to traverse the type hierarchy at runtime (*e.g.*, by using the `isInstance` or `getSuperclass` methods of `Class`). Furthermore, Refined Java requires a large amount of name mangling to support its translations.

In conclusion, we are not convinced that the added complexity of the Refined Java implementation provides any significant benefits. It will be much easier to answer this question once a working Refined Java implementation is available.

8.4 Delayed heterogeneous translation

A heterogeneous translation creates a new class definition for each distinct instantiation of a parametric class. In each class definition, every type variable is replaced by the actual type parameter. For example, in a class `C<T>`, each `T` would be replaced with `String` for the instantiation `C<String>`, while for the instantiation `C<Object>` each `T` would be replaced by `Object`. The generation of the new class definitions can happen at compile time, resulting in multiple `.class` files, or at load time. For the latter technique, the virtual machine's *classloader* (*i.e.*, the part of the JVM responsible for loading class definitions) will generate the new definitions when needed by the running program [AFM97].

Performing heterogeneous translations within the JVM classloader, instead of at compile time, provides many advantages. All desired language features can be supported, including using primitive types as parameters. The only performance costs involves the time needed for the classloader to perform the translations. No casts need to be inserted, and no runtime checks are necessary to ensure safety. A heterogeneous translation, though, does increase the memory footprint of the program, due to the duplication of code for each instantiation.

As the translation is done at load time, this additional space is needed only within the JVM; it does not affect network transmission time.

The main flaw with the implementation of [AFM97] is that it changes the `.class` file format, and is thus incompatible with existing Java Virtual Machine classloaders. This is not a major problem for programs running under a system-level interpreter, as a new classloader can be dynamically added to the system. A new classloader cannot be added to browser-level interpreters, though, due to security restrictions. This method will therefore require users to install new software.

8.5 Modified JVM

The final implementation approach is to change the target platform. Instead of translating parametric Java into standard Java bytecodes, one can modify the Java Virtual Machine to add native support for the new language features.

Obviously, a modified JVM can support all desired language features, including using primitive types as parameters. The specific changes of [MBL97] also allow for efficient structural matching. These changes impose a slight performance cost for standard Java programs; the authors give a figure of 3%. For parametric classes, [MBL97] reports an improvement of up to 17%. [Tho97], though, only achieved an improvement of 5%.

There are three problems inherent in modifying the JVM. First, people need to install the new virtual machine to run the extended programs. This greatly limits the possible acceptance of the language. Also, because the modification require changes to the JVM's bytecode verifier, the security of the modified JVM must be reexamined to ensure that no flaws exist. Finally, debuggers must also be modified to work with the altered JVM.

8.6 A proposal

We feel that the best possible implementation for an extended version of Java is to combine the above approaches. Specifically, a combination of translation, a modified classloader, and a modified JVM can be used to create a single `.class` file that will run on current JVM installations, but gain significant performance improvements when run on the revised JVM.

In this scheme, a compiler would output standard JVM bytecodes, using the `Rupiah` translation methods. The compiler would also add to the `.class` file additional attributes to demarcate what segments of code were compiler generated as part of the translation process. The compiler, for example, would mark as translation artifacts all inserted casts, runtime checks, and operations involving `PolyClass` objects.

A standard JVM will ignore these additional attributes, and execute the `.class` file as if it were a standard Java (or `Rupiah`) program.

The classloader in a modified JVM, though, would recognize the extended attributes, and convert the `.class` file into a form more appropriate to the new JVM. The extended virtual machine would thus be able to execute the `Rupiah` program with increased performance.

Needless to say, this is a broad proposal. Implementing it—especially determining how to best modify the Java virtual machine—will not be a trivial task. We believe, though, that this combination will provide the best tradeoff between performance and backwards compatibility.

Chapter 9

Conclusion

`Rupiah` provides programmers with a powerful set of tools for building safe and reusable classes. To harness the power of `ThisType`, exact types, and match-bounded parametric polymorphism, a programmer must understand how to properly use them. The extra flexibility `Rupiah` provides increases the complexity of the language, when compared to Java or the other proposed extensions. We believe, though, the difficulty of programming in `Rupiah` is not much greater than programming in standard Java, and thus should not hamper the use of the language.

Likewise, we believe that the performance costs imposed by the current `Rupiah` implementation are relatively minor, and therefore will also not affect people's adoption of the language. However, we need to perform more testing and tuning of the implementation. We must also build a library of data structures that make use of `ThisType`, to further evaluate the current implementation, and to improve the ability of programmers to understand how the `Rupiah` extensions should be used.

9.1 Contributions of the thesis

This thesis provides three main contributions. First, we have specified in detail the changes in Java necessary to add `ThisType`, exact types, and match-bounded parametric polymorphism. Unlike the language proposal [Bru97a], we examine and discuss the interaction of these features with such things as casts and the `instanceof` operator.

The second main contribution is the `Rupiah` compiler implementation. The languages `Rupiah` is based on—$\mathcal{LOOM}$ [BFP97] and PolyTOIL [BSvG95]—were implemented through interpreters. Having a compiler-based implementation will allow us to better evaluate the usefulness of the new type constructs, and their possible performance implications.

Finally, we have provided a run-time implementation that targets the standard Java Virtual Machine without compromising the language design. While certain elements of this implementation are based on others' work (*e.g.*, bridge methods and type erasure from [OW97]), the use of `PolyClass` objects and Java's dynamic reflection and introspection capabilities is original work.

9.2 Future work

9.2.1 Current implementation

There are still some important features missing from the current implementation. We do not yet support parameterized methods or inner classes. These features can be supported through translations similar to those used for parameterized classes now (*i.e.*, passing the appropriate `PolyClass` objects as parameters to an inner class constructors, or as extra parameters to a parameterized method). The implementation also needs more testing, as there are certainly still bugs in some of the translations, and possibly the type checker.

A major improvement in performance can be realized by improving the implementation's handling of `PolyClass` objects, so that only one object is constructed for each instantiated type. This would allow the equality of two `PolyClass` objects to be determined in a single operation. It would also reduce the memory demands of `Rupiah` programs.

Interaction with pure Java programs

We would also like to add translations that would allow for `Rupiah` classes to interact with pure Java programs. The first change would allow Java programs to make use of `Rupiah` classes, even if they were not aware of the extended types. This can be implemented through some changes to the constructor translations. We will also define constructors that do not take `PolyClass` objects as parameters. These constructors will initialize the internal `$$class` fields with default `PolyClass` objects, allowing standard Java programs to make use of the classes. However, this translation will only work for parametric classes that do not make use of `ThisType` or exact types. If the class uses `ThisType` or exact types, the runtime parameter checks will likely fail.

The other change would be to allow a programmer to use pure Java collection classes as if they were parameterized `Rupiah` classes. A possible way to implement this is to extend the Java collection class (*e.g.*, `java.util.Vector`) and add the necessary runtime type structures. However, we feel that the time needed to implement this translation would be better spent writing a robust library of collection classes and other vital data structures for `Rupiah`.

9.2.2 Future implementations

An area with much potential for future research is investigating and designing improved implementations for extended Java systems. We would certainly like to see the proposal of Section 8.6 implemented. To do so requires work both in improved JVM designs, and determining the ideal changes to the `.class` file format. The goal, of course, is to minimize the size of compiled `.class` files while still providing sufficient information for a classloader to quickly translate the file into a form suitable for the improved JVM.

The JVM alone provides a wealth of research opportunities. The JVM modifications of [MBL97] are a simple extension (*i.e.*, additional bytecode instructions and data pools). Other modifications might take a more radical approach and alter the design of the JVM to a much larger degree. Possibilities include having a compiler include information about a class that will allow for increased optimizations (*e.g.*, information regarding reference aliasing).

9.2.3 Further language additions

Java's system of access controls—packages—does not scale particularly well to large systems. In particular, the package system does not allow for fine-grained access controls (*e.g.*, limiting access to a field to specific classes). One possibility is to introduce *modules* [Pet96] into the languages. Modules provide a new level of abstraction that further isolates public interfaces from private implementation and allow for fine-grained access control.

Finally, we would like to add *class groups* to `Rupiah`. Class groups capture the best feature of virtual types—allowing one to extend mutually recursive systems of interfaces or classes—while maintaining static type safety. [BOW98] proposes generalizing `ThisType` by allowing the programmer to choose the name of the variable that will represent the interface of `this`. For example, in the interface

```
interface ListIfc (TType)
{
    char head();
    @TType tail();
    void setHead(char h);
    void setTail(@TType t);
}
```

`TType` is the name for `ThisType`. As with `ThisType`, the meaning of `TType` changes in a subinterface; that is, we assume that `TType` extends `ListIfc`.

To capture the functionality of virtual types, we can group together mutually referential interfaces so as to limit the scope of the `ThisType` definitions. This grouping is performed by making use of inner classes. For example, to create a group of interfaces that represent an alternating list, one could write

```
interface ListGrpIfc
{
```

```
  public interface XListIfc (XThis) {
      char head();
      @YThis tail();
      void setHead(char h);
      void setTail(@YThis t);
  }

  public interface YListIfc (YThis) {
      float head();
      @XThis tail();
      void setHead(float h);
      void setTail(@XThis t);
  }
}
```

A class that implements `ListGrpIfc` would have two inner classes to implement `XListIfc` and `YListIfc`.

With this notation, the concept of binary methods is also extended. Within a class group, any method that has a parameter of an extended `ThisType` is considered in a binary method. In `ListGrpIfc`, a method that has a parameter of type `XThis` or of `YThis` is considered a binary method.

Finally, we can add functionality to the alternating list by extending `ListGrpIfc` and the two inner interfaces. For example, to add lengths to the list, we could write

```
interface LengthListGrpIfc extends ListGrpIfc
{
  public interface LenXListIfc (XThis) extends XListIfc {
      int length();
  }

  public interface LenYListIfc (YThis) extends YListIfc {
      int length();
  }
}
```

A class that implements `LengthListGrpIfc` could extend a class that implements `ListGrpIfc` without running into any typing problems.

Bibliography

[AFM97] Ole Ageson, Stephen Freund, and John C. Mitchell. Adding parameterized types to Java. In *ACM Symposium on Object-Oriented programming: System, Languages, and Applications*, pages 49–65, 1997.

[AG96] Ken Arnold and James Gosling. *Java*. Addison Wesley, 1996.

[BCC+95] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[BOSW97] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Generic Java: Java with type parameters. Javasoft, Bell Labs, and University of South Australia, November 1997.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. To appear in ECOOP '98 Proceedings, April 1998.

[Bru97a] Kim B. Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Technical report, Williams College, 1997.

[Bru97b] Kim B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, 1997.

[Bru97c] Kim B. Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. Technical report, Williams College, 1997.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CDG+88] L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report SRC-31, DEC systems Research Center, 1988.

[CJ98] Robert Cartwright and Guy L. Steele Jr. Refined Java. Sun Microsystems Laboratories, April 1998.

[ES90] Margaret A. Ellis and Bjarne Stroustrop. *The annotated C^{++} reference manual.* Addison-Wesley, 1990.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[GRJV94] E. Gamma, R.Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and reuse of object-oriented designs.* Addison-Wesley, 1994.

[Jav97] Javasoft. Inner classes specification. http://www.javasoft.com/products/ jdk/1.1/docs/guide/innerclasses/ spec/innerclasses.doc.html, February 1997.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. M.I.T. Press, Cambridge, MA, 1987.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *ACM Symposium on the Principles of Programming Languages*, pages 132–145. ACM, 1997.

[Mey92] B. Meyer. *Eiffel: the language.* Prentice-Hall, 1992.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *24th ACM Symp. on the Principles of Programming Languages*, pages 146–159. ACM, 1997.

[Pet96] Leaf Petersen. *A module system for LOOM.* Williams College Senior Honors Thesis, 1996.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C^{++}.* Addison-Wesley, 1994.

[Tes85] L. Tesler. Object Pascal report. Technical Report 1, Apple Computer, 1985.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP '97*, pages 444–471. LNCS 1241, Springer-Verlag, 1997.

[Wir71] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

Appendix A

`Rupiah` Grammar

Section numbers correspond to those in [GJS96].

§19.4

ExactType:
    @ ClassOrInterfaceNotExactType

ClassOrInterfaceNotExactType:
    Name Instantiations$_{opt}$

InterfaceNotExactType:
    ClassOrInterfaceNotExactType

ClassOrInterfaceType:
    ClassOrInterfaceNotExactType
    ExactType
    ThisType

Instantiations:
    $<$ ClassOrInterfaceList $>$

ClassOrInterfaceList:
    ClassOrInterfaceType
    ClassOrInterfaceList, ClassOrInterfaceType

§19.8.1

ClassDeclaration:
  Modifiers$_{opt}$ **class** Identifier IfcParamsopt Super$_{opt}$ Interfaces$_{opt}$ ClassBody

IfcParams:
  $<$ IfcParamsList $>$

IfcParamsList:
  IfcParam
  IfcParamList, IfcParam

IfcParam:
  Identifier **extends** InterfaceNotExactType
  Identifier

§19.8.3

MethodDeclarator:
  Identifier IfcParams$_{opt}$ (FormalParameterList$_{opt}$)

§19.8.5

ConstructorDeclarator:
  SimpleName IfcParams$_{opt}$ (FormalParametersList$_{opt}$)

§19.9.1

InterfaceDeclaration:
  Modifiers$_{opt}$ **interface** Identifier IfcParams$_{opt}$ ExtendsInterfaces$_{opt}$ InterfaceBody

§19.12

MethodInvocation:
  Name Instantiations$_{opt}$ (ArgumentList$_{opt}$)

Primary . Identifier Instantiations$_{opt}$ (ArgumentList$_{opt}$)
super . Identifier Instantiations$_{opt}$ (ArgumentList$_{opt}$)

Appendix B

Sample Rupiah Programs

In this appendix we present two examples of Rupiah programs. The first example is a relatively standard example of a polymorphic ordered linked-list class, which can be instantiated using both singly- or doubly-linked nodes. It uses an interface in which the ordering relations are binary methods. The second is a heterogenous ordered linked-list, with a polymorphic upper bound.

B.1 Polymorphic Homogenous Ordered Linked-list

This program defines a class that supports homogeneous linked-lists (*i.e.*, a list in which all elements have exactly the same type).

```
interface ExactNodeIfc<T>
{
  @T getValue();
  void setValue(@T newValue);
  @ThisType getNext();
  void setNext(@ThisType newNext);
}

/*************************************************/

class ExactNode<T> implements @ExactNodeIfc<T>
{
// Fields

    @T value;          // value of node
```

```
    @ThisType next;    // link to next node in list

// Constructors

    ExactNode(@T value) {
        this(value, null);
    }

    ExactNode(@T value, @ThisType next) {
        this.value = value;
        this.next = next;
    }

// Methods

    public @T getValue() {
        return value;
    }

    public void setValue(@T newValue) {
        value = newValue;
    }

    public @ThisType getNext() {
        return next;
    }

    public void setNext(@ThisType newNext) {
        next = newNext;
    }
}

/**************************************************/

interface ExactDblNodeIfc<T> extends ExactNodeIfc<T>
{
    @ThisType getPrev();
    void setPrev(@ThisType previous);
}
```

```
/***********************************************/

class ExactDblNode<T> extends ExactNode<T>
    implements @ExactDblNodeIfc<T>
{
// Fields

    @ThisType previous;  // link to the previous node

// Constructors

    public ExactDblNode(@ThisType previous, @T value, @ThisType next) {
        super(value, next);
        this.previous = previous;
    }

    public ExactDblNode(@T value) {
        this(null, value, null);
    }

// Methods

    public @ThisType getPrev() {
        return previous;
    }

    public void setPrev(@ThisType newPrev) {
        previous = newPrev;
    }

    public void setNext(@ThisType newNext) {
        super.setNext(newNext);
        if (newNext != null) {
            newNext.setPrev(this);
        }
    }
}

/***********************************************/
```

```
interface OrdEltIfc
{
    boolean gt(@ThisType other);
    boolean eq(@ThisType other);
}

/*************************************************/

/* A homogenous ordered list of items of fixed type T that
 * matches OrdEltIfc.  The list could be singly- or doubly-linked
 * depending on the value of N
 */

interface OrdListIfc<T extends OrdEltIfc, N extends ExactNodeIfc<T>>
{
    boolean find(@T match);
    void addNode(@N newNode);
}

/*************************************************/

class OrdList<T extends OrdEltIfc, N extends ExactNodeIfc<T>>
    implements OrdListIfc<T, N>
{
// Fields

    protected @N head;

// Methods

    // Determine if match is contained in the list
    public boolean find(@T match) {
        @N current = head;
        while (current != null && match.gt(current.getValue())) {
          current = current.getNext();
        }
        return current != null && current.getValue().eq(match);
    }

    // Add a node to the ordered list
```

```
    public void addNode(@N newNode) {
        if (head == null) {
            head = newNode;
        } else {
            @N current = head;
            @N prevNode = null;
            while (current != null
                   && newNode.getValue().gt(current.getValue()))
            {
                prevNode = current;
                current = current.getNext();
            }
            if (prevNode == null) {
                newNode.setNext(head);
                head = newNode;
            } else {
                newNode.setNext(current);
               prevNode.setNext(newNode);
            }
        }
    }
}

/***************************************************/

interface SomeOrdIfc extends OrdListIfc { ... }

/***************************************************/

public class SomeOrdClass implements @SomeOrdIfc { ... }

/***************************************************/

public class Test
{
    public static main(String[] args) {
        // Singly-linked homogenous list
        OrdListIfc<SomeOrdIfc, ExactNodeIfc<SomeOrdIfc>> slist
            = new OrdList<SomeOrdIfc, ExactNodeIfc<SomeOrdIfc>>();
```

```
        // Doubly-linked homogenous list
        OrdListIfc<SomeOrdIfc, ExactDblNodeIfc<SomeOrdIfc>> dlist
            = new OrdList<SomeOrdIfc, ExactDblNodeIfc<SomeOrdIfc>>();

        @SomeOrdIfc elt = new SomeOrdClass( ... );
        slist.addNode(new ExactNode<SomeOrdIfc>(elt));
        if (slist.find(elt)) { ... }

        dlist.addNode(new ExactDblNode<SomeOrdIfc>(elt));
        if (dlist.find(elt)) { ... }
    }
}
```

B.2 Polymorphic Heterogeneous Ordered Linked-list

This program supports heterogeneous ordered linked-lists, where the user may choose the upper bound of the elements in the list. The elements must match the interface `HetOrdEltIfc<T>`. Note that this example requires the use of F-bounded matching in order to capture the polymorphic constraints on the heterogeneous lists.

```
interface ThisNodeIfc<T>
{
    T getVal();
    @ThisType getNext();
    void setVal(T newVal);
    void setNext(@ThisType newNext);
}

/**************************************************/

class ThisNode<T> implements @ThisNodeIfc<T>
{
// Fields

    protected T val;           // stored element
    protected @ThisType next;  // link to next node

// Constructors

    public ThisNode(T value, @ThisType next) {
```

```
      val = value;
      this.next = next;
    }

    public ThisNode(T value) {
      this(value, null);
    }

// Methods

    public T getVal() {
      return val;
    }

    public @ThisType getNext() {
      return next;
    }

    public void setVal(T newVal) {
      val = newVal;
    }

    public void setNext(@ThisType newNext) {
      next = newNext;
    }
}

/***************************************************/

interface ThisDblNodeIfc<T> extends ThisNodeIfc<T>
{
    @ThisType getPrev();
    void setPrev(@ThisType newPrev);
}

/***************************************************/

class ThisDblNode<T>  extends ThisNode<T> implements @ThisDblNodeIfc<T>
{
// Fields
```

```
    protected @ThisType prev;  // Link to previous node

// Constructors

    public ThisDblNode(T val, @ThisType prev, @ThisType next) {
      super(val, next);
      this.prev = prev;
    }

    public ThisDblNode(T val) {
      this(val, null, null);
    }

// Methods

    public @ThisType getPrev() {
      return prev;
    }

    public void setPrev(@ThisType newPrev) {
      prev = newPrev;
    }

    public void setNext(@ThisType newNext) {
      super.setNext(newNext);
      if (newNext != null) {
        newNext.setPrev(this);
      }
    }
}

/*************************************************/

/* A heterogenous ordered list of items all matching T,
 * where T extends HetOrdEltIfc<T>. It could be singly-
 * or doubly-linked, depending on the value of N
 */

interface OrdListIfc<T extends HetOrdEltIfc<T>,
```

```
     N extends ThisNodeIfc<T>>
{
    boolean find(T match);
    void addNode(@N newNode);
}

/***************************************************/

class OrdList<T extends HetOrdEltIfc<T>,
      N extends ThisNodeIfc<T>>
implements OrdListIfc<T, N>
{
// Fields

    protected @N head;  // head of list

// Constructors

    public OrdList() {
      head = null;
    }

// Methods

    public boolean find(T match) {
        @N current = head;
        while (current != null && match.gt(current.getVal())) {
            current = current.getNext();
        }
        return current != null && current.getVal().eq(match);
    }

    public void addNode(@N newNode) {
        if (head == null) {
            head = newNode;
        } else {
            @N current = head;
            @N prevNode = null;

            while (current != null
```

```
                 && newNode.getVal().gt(current.getVal()))
            {
                prevNode = current;
                current = current.getNext();
            }
            if (prevNode == null) {
                newNode.setNext(head);
                head = newNode;
            } else {
               newNode.setNext(current);
               prevNode.setNext(newNode);
            }
        }
    }
}

/****************************************************/

interface HetOrdEltIfc<T>
{
    boolean gt(T other);
    boolean eq(T other);
}

/****************************************************/

interface SomeOrdIfc extends HetOrdEltIfc<SomOrdIfc>
{
    ...
}

/****************************************************/

class SomeOrdClass implements SomeOrdIfc { ... }

/****************************************************/

public class Test
{
    public static void main(String[] args) {
```

```
        OrdListIfc<SomeOrdIfc, ThisNodeIfc<SomeOrdIfc>> slist
            = new OrdList<SomeOrdIfc, ThisNodeIfc<SomeOrdIfc>>();

        OrdListIfc<SomeOrdIfc, ThisDblNodeIfc<SomeOrdIfc>> dlist
            = new OrdList<SomeOrdIfc, ThisDblNodeIfc<SomeOrdIfc>>();

        SomeOrdIfc elt = new SomeOrdClas( ... );

        slist.addNode(new ThisNode<SomeOrdIfc>(elt));
        if (slist.find(elt)) { ... }

        dlist.addNode(new ThisDblNode<SomeOrdIfc>(elt));
        if (dlist.find(elt)) { ... }
    }
}
```

Appendix C

Running the Rupiah Compiler

The Rupiah compiler is written in Java, and thus should run on any platform that has a Java 1.1 runtime. At this point, we have only run the compiler on Sun UltraSparc workstations running Solaris 2.5.1, using version "JDK1.1.3/jws:97.08.28" of Sun's Java Development Kit.

The Rupiah compiler distribution is available through anonymous FTP:

`ftp://ftp.cs.williams.edu/pub/kim/Rupiah/`

The binary version of the compiler is contained in the file `rupiah.zip`. This file is an archive of the compiler's classes. Also in this directory is the file `PolyClass.class`, which must be available to all Rupiah programs. This file is included in `rupiah.zip`—the separate version is intended for people who wish to distribute Rupiah programs.

To run the compiler, the file `rupiah.zip` must be part of the current classpath. The command

```
java -ms8m EDU.williams.tools.javac.Main
```

will execute the compiler. The command line syntax is identical to that of `javac`, Sun's Java compiler.

As of the printing of this document, the compiler does not implement parameterized methods or inner classes, and per-instantiation static variables. Also, the compiler does not support separate compilation for Rupiah classes: the source files for all referenced classes must be included in the compilation command. For example, to compile a class `C` that makes use of the class `Vector<T>`, the following command line should be used:

```
java -ms8m EDU.williams.tools.javac.Main C.java Vector.java
```

Support for these features is forthcoming.

As the compiler is based on Sun's `javac` compiler, we cannot make the source code available to the general public.