`Rupiah`: Towards an Expressive Static Type System for Java

by

John N. Foster

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 20, 2001

Abstract

Despite Java's popularity, several practical limitations imposed by the language's type system have become increasingly apparent in recent years. A particularly glaring omission is the lack of a generic mechanism. As a result of this shortcoming, many recent projects have extended Java to support polymorphism in the style of C++ templates or Ada generics. One project, GJ [BOSW98], adds F-bounded parametric polymorphism [CCH+89] to Java via a homogeneous translation (such that only one class file results from each compiled source file), and produces bytecode that is compatible with the standard Java Virtual Machine. However while GJ's simple translation based on erasure allows for maximum interaction with existing Java code, the new parameterized types that it supports do not operate consistently with Java's semantics for lightweight reflection (*i.e.*, checked type-casts and `instanceof` operations).

We present `Rupiah`, a language based on features adapted from LOOM [BFP97], a provably type-safe language, and implemented by a translation based on GJ. However its translation differs from GJ's in that it harnesses Java's built-in reflection to store information about parameterized types. The resulting bytecode correctly executes checked cast and `instanceof` expressions because it has access to the necessary type information at run-time. We also add a `ThisType` construct, which solves many of the problems that arise when binary methods are mixed with inheritance, and we replace subtyping with a different relation, matching. Finally, we add exact types, an inheritable virtual constructor mechanism: `ThisClass`, and compiler features to allow separate compilation of `Rupiah` source files. These features are implemented in a modified `javac` compiler. Bytecode emitted by our compiler runs on any Java 1.2 VM. Thus, the `Rupiah` project contributes a complete implementation of an extension of Java with a more expressive type system that maintains a close fit with existing Java semantics and philosophies.

Acknowledgements

This thesis would have been impossible to complete without the guidance of one person. It's hard to imagine an advisor with a better mix of knowledge, encouragement, and patience than Kim Bruce. Thank you Kim for pushing me to take on this project and for having the patience to help me through it.

This work directly builds on the efforts of Jon Burstein. His research tackled a large portion of the challenges of specifying and implementing `Rupiah`. Every day that I worked on this project, I benefited from Jon's previous efforts.

I also would like to thank Tom Murtagh who as my second reader provided invaluable advice and insights along the way. I'd like to thank Jay Sachs for teaching such an excellent course in the spring of 1999 that first got me interested in programming languages. Also thank you to Barbara Lerner, Williams' resident object-oriented design guru, for helping me to work through some difficult program designs as well as volunteering (!) to read drafts.

Thanks to everyone who made the many long days and nights in Schow and the UNIX lab fun, especially Art Munson '01, Jesse Davis '02, Chris Kelley '03, Shimon Rura '03, Sean Sandys '94, and Duane and Mary Bailey.

Finally, thank you to my friends and family for their continued love and support. Especially thanks to my parents in Ann Arbor, Phil and Sharon, and in Seattle, Kent and Susanne, as well as my sisters: Clare, Mollie, and Ella. Lastly, thanks to my close friends for putting up with me this year: especially Cristie, Abhaya, Alex, Daren, John, JJ, Krzys, Lynn, Pete, and Xavi.

Contents

List of Figures

Chapter 1

Introduction

While programming languages have evolved towards higher-level languages, programming itself is still often an error-prone task. Over time, issues of safety and confidence in the correctness of programs have become more important to programmers. Typed languages come with a guarantee that certain type errors will not occur in legal programs. For example, in an object-oriented language, a type system can guarantee that a type-checked method call will never fail because the method is not defined. In this thesis, we identify some limitations in the type system of the Java[1] programming language [GJS96]. We propose several new type abstractions that address these problems and allow for a greater degree of protection for programmers. The incorporation of these new features allows early detection of errors for more programs, and greater safety for programmers.

1.1 Type Systems in Programming Languages

Programming language designers develop type systems to protect and aid programmers. Over time, programmers have embraced high-level languages that contain stronger and more expressive type systems because of the advantages they provide. Additional structure beyond the syntax of a language allows a higher degree of confidence in the correctness of a program written in the language. Most modern programming languages are *strongly typed*, meaning that every expression in the language has a type that is checked by the type checker. Many are also *statically typed*, meaning that the type checking happens at compile time. Alternatively, a strongly typed language may be *dynamically typed*, meaning the type checking happens when programs are run. In this thesis, we are primarily concerned with

[1] Java is a trademark of Sun Microsystems.

statically typed languages.

Static type systems are useful because they alert the programmer at an early stage in the development cycle to the possibility of undesirable behavior in a program. For example, in object-oriented languages we would like to have the guarantee that a method call, if type checked, can always be safely invoked at run-time. Instance methods are always invoked on objects so we must ensure that the method is defined for each object that it is called on. A type system can provide such a guarantee. For example, consider the following method (written in Java syntax):

```
public static Integer add(Integer i1, Integer i2) {
  int i1val = i1.intValue();
  int i2val = i2.intValue();
  return new Integer(i1val + i2val);
}
```

This method adds the numeric values of two `Integer` objects, and returns an `Integer` object that contains their sum. Inside the body of this method, we invoke the method `intValue()` on both arguments. Therefore, we would like to be sure that these methods can actually be called on the actual parameters at run-time.

In a language with no type system, we could ignore the declared argument types and call `add` with any two objects as arguments. For example, we could invoke `add` with one parameter of type `Integer` and one of type `String` as follows:

```
add(new Integer(1), ``foo'');
```

However, in executing that invocation, an error occurs in the body of `add` when we try to invoke the method `intValue()` on ``foo'' because the `intValue()` method is undefined for objects of type `String`. Thus, an error occurs in the program at run-time.

In contrast, a language with a type system can ensure that any object passed to `add` will have an `intValue()` method defined. The error illustrated above does not occur because the type system does not allow a `String` to be passed as the second argument to ``foo''. A type system therefore protects the programmer from running a program with such undesired effects. It provides a guarantee that any program accepted by the type system will not suffer from certain catastrophic blowups.

Static type systems provide a strong assurance of protection for the programmer because type checking happens at compile time. Because the type checking is done before the program is run, errors are detected earlier rather than later. More importantly, a program can only be compiled if it is a legal program according to the type-checking rules. In order to implement a static type system, the compiler includes a type checking phase that analyzes each statement in a source program and emits an error if the statement is not type safe according to the information available to the compiler.

As noted earlier, an alternative to a static type system is a dynamic one. In a language with dynamic type checking, statements are not checked at compile time, but instead as they execute. A statement which fails to pass the type checker results in a run-time exception. Dynamic type systems provide a different kind of type safety because errors may not become evident until long after compilation.

1.2 Problems With Type Systems

The evolution of strong yet expressive type systems for programming languages has not proceeded without difficulty. Type systems are restrictive by design; a type system disallows programs that would otherwise be accepted by the grammar of the language. Thus a programming language that includes a type system has fewer legal programs than the language generated by the grammar alone. An optimal type system for a language is one that only makes erroneous programs illegal yet still provides a very strong guarantee of safety for the programs it accepts. The *expressiveness* of a type system characterizes the degree to which abstract concepts can be represented within the bounds of the system. Usually, increasing the expressiveness of a type system, by making it capable of representing more complex concepts, increases the number of useful programs that are legal.

Often type systems are flawed because they are too weak or too rigid, or simply not expressive enough to write many useful programs. Pascal is a language with an inflexible type system. Because Pascal's type system is very strong, it provides a strong measure of confidence in the programs that it accepts. However, Pascal's type system is not very expressive. Many useful programs are impossible to write because the type system is so rigid.

The *C* and C^{++} programming languages have more flexible type systems than Pascal because they allow the programmer to override a declared type by inserting a cast. Still, the static type systems of *C* and C^{++} are not expressive enough, frequently forcing the programmer to circumvent the type system with type casts. Thus while they have strong, static type systems, *C* and C^{++} do little in practice to protect programmers. Any practical benefit gained from the presence of a type system is negated by the fact that in most programs, the system must be circumvented.

1.3 Type System issues in Object-Oriented Languages

Object-oriented languages have experienced great success in recent history. One of the reasons for their success is the increased potential for code reuse in the object-oriented paradigm. Java [GJS96] improves on other popular languages such as C^{++} by incorporating concepts such as automatic memory management via garbage collection, cross-platform compatibility, and a powerful security model for programs distributed over networks. How-

ever, like many object-oriented languages, Java's type system suffers from many problems that have been solved in traditional imperative and functional languages languages. Object-oriented languages have been slow in general, to adopt recent advances such as polymorphism and static error detection [Car96]. We identify two main weaknesses in Java as particularly troublesome: lack of a generic mechanism and problems with binary methods, both described in detail in Chapter 2.

1.4 Initial Goals

The top-level goal of the `Rupiah` project is to increase the expressiveness of Java's type system. Our hope is that with a more expressive type system, many more programs can be written safely within the bounds of the static type system. Thus, we present a preliminary set of goals for this thesis. Not all of these properties are simultaneously achievable, instead this list provides a set of optimal properties of our language. A more detailed set of goals are described in Section 3.5.

- **New Expressive Features:** We wish to add features to significantly increase the expressiveness of Java. These include: parametric polymorphism (Section 2.2), `ThisType` (Section 2.4), and `ThisClass` constructors (Section 5.2).
- **Java Compatibility:** Any existing Java source programs should be legal `Rupiah` programs. Compiled `Rupiah` class files should execute on the existing Java virtual machine (JVM). Also, existing compiled classes should be usable by `Rupiah` classes.
- **Transparent Translation:** The design of the language should be completely independent of the translation to bytecode. That is, the programmer should not be aware of the translation when writing `Rupiah` programs.
- **Run-Time Types, Reflection:** Type expressions in the bytecode generated by the `Rupiah` compiler should operate correctly in Java's introspective operations. These include:
  - `instanceof` expressions
  - cast expressions
  - `new` array expressions
  - Java's reflection API.
- **Reasonable Efficiency:** Our implementation should have the property that standard Java programs compiled with our compiler are as efficient as if they were compiled with `javac`. Secondly, the extra performance and space requirements for programs that use new features should be minimal (some discussion of what is reasonable is given in Section 3.5).

1.5 Acknowledgements

Many of the ideas presented in this thesis are adapted from the work of others. A series of languages developed at Williams College under Kim Bruce: TOOPLE[Bru93], **TOIL** [BvG93], **PolyTOIL** [BSvG95], and $\mathcal{LOOM}$ [BFP97] provided the concepts for many of the type system features we propose adding to Java. A paper by Bruce [Bru97] first proposed adding match-bounded parametric polymorphism, `ThisType`, and exact types to the language. These features were the subject of an honors thesis by Jon Burstein '98 [Bur98] and most of them were implemented in a compiler. Thus, this work directly builds on the results of several preceding languages and projects.

1.6 Thesis Overview

The first section of this thesis provides some background, introduction, and motivations for the new features included in Rupiah. Chapter 2 describes some type problems with object-oriented programming languages, including Java. The two main problems are the lack of a generic mechanism, and problems with binary methods. It also proposes some solutions to those problems: parametric polymorphism, and `ThisType` respectively. Chapter 3 describes some other extensions of Java that have explored various ways to translated advanced type system features to Java bytecode. We primarily look at GJ [BOSW98] and NextGen [CS98].

The second section describes `Rupiah`. Chapter 4 gives a detailed look at the implementation of core `Rupiah` features: parametric polymorphism, `ThisType`, and exact types. Chapter 5 describes the new features that we add in this thesis: polymorphic methods and inner-classes, a `ThisClass` constructor, and bytecode `attributes` to support separate compilation.

The final section begins with Chapter 6 which compares and evaluates the various proposals for adding new type system features to Java. We consider both the expressiveness gained from these features, and the implementation of each language. Chapter 7 describes some ideas for future work in the `Rupiah` project. In the appendices, we give a grammar for `Rupiah`, some sample programs implemented in `Rupiah`, and some practical details about the `Rupiah` compiler.

Chapter 2

Motivations

Here we describe two problems in Java that have motivated the development of `Rupiah`. Firstly, the lack of a generic mechanism in Java makes many programs difficult to write within the static type system. Secondly, binary methods – methods that take a parameter of the same type as the object that they are invoked on – cause problems in languages with inheritance. We describe these problems using a list data structure as an example and propose type-system solutions to them: parametric polymorphism, `ThisType`, and exact types.

2.1 Genericity

Java's type system's most oft-cited weakness is the lack of a generic mechanism. Generic types are an important language feature, especially for programming certain structures known as container classes. A container class is a structure such as a list, stack, or queue that holds collections of values in an organized way. The difficulty of writing container classes in Java stems from the fact the language lacks a mechanism to write code where the types of objects stored in the container are abstracted.

Suppose that we wish to hold some integers and strings in separate list-like structures in a Java program. For our example, we'll consider a list to be a structure that supports the following four operations:

- `new List()` – initializes an empty list
- `add(data)` – adds `data` to the beginning of the list

- `remove()` – removes the first element of the list and returns it. If the list is empty, throw an exception.
- `empty()` – returns true if the list is empty, false otherwise.

A standard way to implement this kind of list is to construct it out of list nodes. A list node is a simple structure that contains fields representing the data stored in that node and a link to the next node in the list.

A naive approach to the problem is to write two distinct classes: one a list of integers, and one a list of strings as shown in Figures 2.1 and 2.2. This solution satisfies the bare requirements of the problem, but is less than ideal because it requires duplication of code. The source definitions and programming logic for each are identical except for the instances of `String` and `Integer`. One could create the `IntegerList` class from the `StringList` class definition simply by doing a manual replacement of `String` with `Integer` and `IntegerList` with `StringList`. This needless duplication of code is undesirable both for the increased initial effort and much greater cost of maintaining the code. Further, future code-reuse is severely hindered with this approach.

Another way to solve the problem is to write a more generic `List` that can hold any data type we might need to store. This technique for programming container classes holds objects of the most generic type that the language supports. In *C* or *C*++, this corresponds to list elements of type `*void`, in Java the element types are `Object`. An example using this technique is shown in Figure 2.3. This implementation of `List` takes advantage of *subtype polymorphism* – a feature common to most object-oriented languages. Subtype polymorphism in Java simply means that an object may be assigned to, or used as a formal parameter whose declared type is a supertype.[1] In our `List` example, by declaring the storage type as `Object`, we allow any object type to be stored in the list, as all objects in Java are subtypes of `Object`.

This technique seems to accomplish what we want: a container class that can store values of any type without needless duplication of code. Because the storage type of the `List`'s elements is `Object`, the compiler easily determines that any object type can be stored in the list. However, because the return type of `remove()` is statically declared as `Object`, the only type information available at compile time about elements removed from the list is that their type is `Object`. This is problematic because explicit type conversions (casts) must be inserted, as shown in Figure 2.4, whenever an element is removed from the list in order to pass the type checker. The act of adding and then removing an object from the list loses all static type information about that object.

Using a cast to correct this problem is highly undesirable for several reasons. First, it requires extra program code which can lead to errors and increase the cost of maintaining the program. Secondly, a cast is actually a circumvention of the static type system; thus

[1] See the footnote in Section 2.4 on why Java does not truly support subtype polymorphism.

```
public class EmptyListException extends Exception { }

public class StringNode {
  protected String data;
  protected StringNode next;

  public StringNode(String data, StringNode next) {
    this.data = data;
    this.next = next;
  }
  public String getData() { return data; }
  public StringNode getNext() { return next; }
  public void setNext(StringNode newNext) { next = newNext; }
}

public class StringList {
  protected StringNode head;

  public StringList() { head = null; }
  public void add(String data) {
    head = new StringNode(data, head);
  }
  //pre: list is not empty
  public String remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    StringNode oldHead = head;
    head = head.getNext();
    oldHead.setNext(null);
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 2.1: Separate list for `String`

```
public class IntegerNode {
  protected Integer data;
  protected IntegerNode next;

  public IntegerNode(Integer data, IntegerNode next) {
    this.data = data;
    this.next = next;
  }
  public Integer getData() { return data; }
  public IntegerNode getNext() { return next; }
  public void setNext(IntegerNode newNext) { next = newNext; }
}

public class IntegerList {
  protected IntegerNode head;

  public IntegerList() { head = null; }
  public void add(Integer data) {
    head = new IntegerNode(data, head);
  }
  //pre: list is not empty
  public Integer remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    IntegerNode oldHead = head;
    head = head.getNext();
    oldHead.setNext(null);
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 2.2: Separate list for `Integer`

```
public class Node {
  protected Object data;
  protected Node next;

  public Node(Object data, Node next) {
    this.data = data;
    this.next = next;
  }
  public Object getData() { return data; }
  public Node getNext() { return next; }
  public void setNext(Node newNext) { next = newNext; }
}

public class List {
  protected Node head;

  public List() { head = null; }
  public void add(Object data) {
    head = new Node(data, head);
  }
  //pre: list is not empty
  public Object remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    Node oldHead = head;
    head = head.getNext();
    oldHead.setNext(null);
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 2.3: `List` with element type `Object`

```
List intList = new List();   //List of Integers
List strList = new List();   //List of Strings

intList.add(new Integer(42));
strList.add(``Java'');

Integer i = (Integer) intList.remove();
String  s = (String) strList.remove();
```

Figure 2.4: Removing elements from `List` defined in Figure 2.3

the nice properties of static type safety for the statement that calls `remove()` are lost. In Java, a cast passes the static type checker except when it can be proved that the cast can never succeed (*e.g.*, a cast from unrelated types such as `Integer` to `String`) [GJS96]. Therefore, by using a cast in Java, the programmer loses some certainty of the correctness of their program because they circumvent the static type system and defer some of the type checking until run-time. Lastly, type casts in this context are undesirable in a broader sense because homogeneous lists can be enforced with a more expressive type system. That is, we'd like the ability to specify that `intList` may only store `Integers` and `strList` may only store `Strings`. Then, storing a `String` in a `intList` or an `Integer` in a `strLst` is a compile-time error. Also, if we retrieve an element from `intList` we know that the element must have type at least `Integer` and similarly `String` for `strList`. However, this kind of behavior is not possible within the bounds of Java's static type system unless we use the technique of Figure 2.1 and 2.2.

2.2 Adding Genericity with Parametric Polymorphism

A solution to the problems illustrated above is *parametric polymorphism* – a type system feature that allows types to be passed as parameters. With parametric polymorphism, type variables and parameterized types may be used in any type expression. This abstraction is a powerful tool that allows container classes to be easily written in a generic way that solves the problems described previously.

2.2.1 Parametric Polymorphism Syntax and Semantics

In Figure 2.5, we define a parameterized class `List<T>`. First we introduce a type variable `T` that stands for the type of the elements in the list. We declare type variables for a class with the `<>` brackets, syntax borrowed from C^{++}. We may then use `T` as a valid type anywhere inside the class. We can also use parameterized types, such as `Node<T>`, inside the class.

```
public class Node<T> {
  protected T data;
  protected Node<T> next;

  public Node(T data, Node<T> next) {
    this.data = data;
    this.next = next;
  }
  public T getData() { return data; }
  public Node<T> getNext() { return next; }
  public void setNext(Node<T> newNext) { next = newNext; }
}

public class List<T> {
  protected Node<T> head;

  public List() { head = null; }
  public void add(T data) {
    head = new Node<T>(data, head);
  }
  //pre: list is not empty
  public T remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    Node<T> oldHead = head;
    head = head.getNext();
    oldHead.setNext(null);
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 2.5: `List<T>` using parametric polymorphism

```
public interface Comparable<T> {
  public int compareTo(T other);
}

public class SortableList<T extends Comparable<T>> extends List<T> {
  public SortableList(T head, SortableList<T> tail);
  public void sort() { ... }
}
```

Figure 2.6: Adding bounds to type parameters

When a polymorphic class is used, its type parameters must be *instantiated* with actual types. For example, to create a list of strings we instantiate the type parameter `T` with the actual type `String`. We write:

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

Instantiating a type variable `T` with a specific type means that each type `T` now has the type of its instantiation. For example, if `listOfString` is declared as above, then the expression:

```
listOfString.remove()
```

has type `String` because `remove` has declared type `T`, and `T` is instantiated with `String`. We can also catch errors such as inserting a `String` into a `List<Integer>`, because `add` takes a parameter of type `T`, which is `Integer` for `List<Integer>`.

Thus, implementing the list structure in a language with a more expressive type system that includes parametric polymorphism allows the programmer to write container classes that can store elements of any type and eliminates the need for casts when accessing elements. This is possible because the compiler is able to statically analyze the instantiations of type parameters.

As declared in Figure 2.5, `T` may be instantiated with any object type. Most flavors of polymorphism allow the programmer to restrict the types that may be used to instantiate a type variable. The *bound* of a type parameter constrains the types that may be used to instantiate it. In a Java-like language that supports parametric polymorphism, we can declare a sortable list as shown in Figure 2.6 by setting a bound on the type variable and adding a `sort()` method. Bounds on type parameters are specified with the `extends` (or `implements`) keyword.

We constrain the element types that can be used to instantiate a `SortableList<T>` by bounding `T` with `Comparable<T>`. Any type which is used to instantiate `SortableList<T>` must satisfy the bound `Comparable<T>` – that is, it must `extend` or `implement` `Comparable<T>`. The bound on `T` makes the following declaration illegal:

```
SortableList<Object> slo = new SortableList<Object>(null, null);
```

because `Object` does not satisfy the bound `Comparable<T>`. The constraint ensures that each element has a `compareTo` method, and thus that some ordering can be used to sort the list elements.

F-bounded Polymorphism

F-bounded polymorphism [CCH+89] refers to a specific kind of parametric polymorphism that allows type variables to be used in the specification of bounds for other type variables. For example, we might declare a list that is generic in both the types of elements that it holds and the nodes used to represent the elements. We write the declaration for this class as:

```
public class List<T, N extends Node<T>>
```

Note in particular that `T` is used in the declaration of the bound of `N` – `Node<T>`. F-bounded polymorphism is a powerful feature that allows type variables to be constrained with very specific types. Problems with binary methods, discussed in Section 2.3 prevent us from writing `List<T, N>` with simple F-bounded polymorphism. Only once these problems are resolved can we write them safely. Thus, the definition of `List<T, N>` is postponed until Section 2.4.

2.2.2 Parametric Polymorphism Summary

Problems writing container classes in Java have suggested that a generic mechanism should be added to the language. Parametric polymorphism, a technique developed in many other procedural, functional, and even object-oriented languages, is a standard solution to the problem and fits cleanly with Java.

2.3 Binary Methods

A second set of problems with Java's type system are the complications that stem from binary methods. A *binary method* is a method that takes a parameter of the same type as the receiver of the method call. Problems with binary methods are common to many object-oriented languages and are explored in detail in [BCC+95].

In general, the difficulty is that binary methods cause problems in a language with inheritance (*i.e.*, most object-oriented languages). When a binary method is inherited by a subclass, the type of the parameter remains the type of the class that originally declared it. The programmer would like the inherited method to still behave like a binary method. However in Java, the method ceases to be a true binary method in the subclass because it

accepts arguments whose type differs from its own. Most object-oriented type systems do not explicitly support the concept of binary methods. However binary methods naturally occur in many programs, including most recursive data structures. Therefore, a truly expressive type system must be able to express binary methods.

Extending our `List<T>` example to work with doubly-linked nodes illustrates the binary method problem directly. In writing `DblNode<T>`, we would like to inherit some of the code from `Node<T>` because the programming logic of the common methods is essentially the same. For example, we should not have to rewrite the `getData()` method. Thus, we define `DblNode<T>` as shown in Figure 2.7. However, this causes problems with the type system. The method `setNext` in `Node<T>` is a binary method – that is, it takes a parameter of type `Node<T>`. Because of the way that we defined `DblNode<T>`, as well as having a `setNext` method with declaration:

```
public void setNext(DblNode<T> newNext)
```

it inherits the `setNext` method from `Node<T>` that takes a parameter of type `Node<T>`. This is problematic because we can call `setNext` to set the `next` link of a doubly-linked node to a singly-linked node. However, we do not wish to insert a `Node<T>` into a chain of `DblNode<T>`, because we expect to be able to call `setPrev` on the nodes in a doubly-linked list.

A related problem is that `getNext()` has return type `Node<T>`, while for a doubly-linked node it should have return type `DblNode<T>`. This problem with return-types is actually somewhat unique to Java. Many object-oriented languages allow methods in subclasses to override methods while making covariant changes in return types. However, Java does not allow this kind of change, even though it is type-safe. Still, this problem is related to the binary method problem, because it illustrates the difficulties in expressing a type that is the same as the type of `this`.

The crux of the binary method problem in this example is that the `setNext` method which only took a parameter of the same type as the object that it is invoked on, does not have that property when inherited by a subclass. Of course, we can write `DblNode<T>` and `Node<T>` as unrelated classes, but then we must re-write much of the code for `DblNode<T>`. Alternatively, we can override the method with signature

```
void setNext(Node<T>)
```

in `DblNode<T>` so that it throws an exception if called with a parameter of type `Node<T>` as shown in Figure 2.8, but then we lose static type safety, because we depend on exceptions to compensate for a weakness in the type system.

```
public class DblNode<T> extends Node<T> {
  protected DblNode<T> prev;

  public DblNode(T data, DblNode<T> next, DblNode<T> prev) {
    super(data, next);
    this.prev = prev;
  }
  public void setNext(DblNode<T> newNext) {
    next = newNext;
    if (newNext != null) newNext.prev = this;
  }
  public void setPrev(DblNode<T> newPrev) {
    prev = newPrev;
    if (newPrev != null) newPrev.next = this;
  }
  public DblNode<T> getPrev() { return prev; }
}
```

Figure 2.7: Definition of `DblNode<T>`

```
void setNext(Node<T> newNext) {
  if (!(newNext instanceof DblNode<T>))
     throw new RuntimeException(``newNext not a DblNode<T>'');
  else
     setNext((DblNode<T>) newNext);
}
```

Figure 2.8: Defining `DblNode<T>`'s `setNext` method

2.4 Adding `ThisType` to support Binary Methods

In Java, the keyword `this` is a reference to the currently executing object. Therefore, a `this` expression in a method can actually refer to a different class than the one that defined it if the method is inherited by a subclass. However, while Java provides a `this` expression that changes reference in subclasses, it does not provide a type that changes meaning in a parallel fashion. `ThisType` gives exactly this functionality. It essentially provides a mechanism for referring to the interface type of `this`. The key feature of `ThisType` is that it changes meaning when program code containing a `ThisType` type expression is inherited by subclasses. Thus, if a method in a class `C` contains an expression with type `ThisType`, then that expression has the type of `C`'s interface when executed by an instance of `C`. However, if the method is inherited by a subclass, `D`, then the same expression with type `ThisType`, has the type of `D`'s interface.

This solution to the binary method problem in object-oriented languages is described in [BCC+95] and implemented in **TOOPLE**[2] [Bru93], **PolyTOIL** [BSvG95], and $\mathcal{LOOM}$ [BFP97]. `ThisType` solves the binary method problem by supporting the naming of the type of `this` in the type system. A `ThisType`-like construct also exists in the Eiffel [Mey92] and Trellis/Owl [SCB+86] languages.

Using `ThisType`, we can easily redefine our node implementations so that `setNext` is truly a binary method, as shown in Figure 2.9. Rather than using class types for the types of parameters in binary methods, we can use `ThisType` to avoid the problems introduced with mixing binary methods and inheritance. Because the meaning of `ThisType` changes when code is inherited to a subclass, we can specify that a method must always takes a parameter of the same type as the receiver of the method. Our `setNext` method as defined with a parameter of type `ThisType` can only be invoked on objects with the same type as `this`. Thus, the behavior of `ThisType` is exactly what is needed to write `setNext` in a statically type safe way. Even when the method is inherited by a subclass, the method continues to be a binary method because of the implicit type change that `ThisType` allows.

With proper support for binary methods in the nodes, we can define a list that is generic in the type of data that it stores and the type of node used to represent the list structure. The definition for `List<T, N>` is shown in Figure 2.10. By making the types of nodes generic, we can use either node implementation internally in our list. For example, a list that uses `DblNode<T>` internally can be declared:

```
List<T, DblNode<T>> dblList = new List<T, DblNode<T>>();
```

There is one problem with our implementation of `List<T, N>`. The assignment statement in `add`:

```
head = new Node<T>(newData, head);
```

[2] `ThisType` is called *MyType* in **TOOPLE** and $\mathcal{LOOM}$

is illegal because `head` has type `N`, while the new instance expression on the right has type `Node<T>`. If `N` is actually instantiated with `DblNode<T>` for example, this would cause a problem as this statement assigns a superclass, `Node<T>`, to an expression whose type is a actually a subclass, `DblNode<T>`. A full description of the problems of constructing objects whose type is a parameter, as well as a solution, are explored in Chapter 5.

2.4.1 Matching

Unfortunately, the addition of `ThisType` introduces a possible hole into our type system. In a language with a `ThisType`-like construct, subclasses do not always generate subtypes. Informally, a subtype is a type that may be used anywhere that a supertype is expected. The following method demonstrates why `DblNode<T>` is not a subtype of `Node<T>`:

```
void breakit(Node<T> node1, Node<T> node2) {
     node1.setNext(node2);
}
```

If we assume that subclasses always generate subtypes, and invoke `breakit` with an argument of type `DblNode<T>` as the first parameter and a `Node<T>` as the second parameter, then a type error occurs. Because `node1` has type `DblNode<T>`, its `setNext` method – that has a single parameter of type `ThisType` – may only be passed a parameter that has the interface type of `DblNode<T>`. Passing `setNext` an argument of type `Node<T>`, as in this example, results in a type error. Therefore, we can not use a `DblNode<T>` wherever a `Node<T>` is legal – that is, it is not a subtype of `Node<T>`.

To solve this problem, we replace subtyping based on subclasses by a slightly weaker relation, "matching," as defined in [BFP97]. In $\mathcal{LOOM}$ an interface type `J` matches another interface type `I` if and only if for every method defined in `I`, a method with the same name and type signature exists in `J`, treating `ThisType` as the same type in the two interfaces. However, because Java does not in general define relationships between types based on structural properties, we revise this relation slightly. Thus we say that an interface type `J` *matches* `I` if and only if one of the following is true:

- `J` is `I`
- `J` extends `I`
- `J` extends `K` and `K` matches `I`

Note that this is essentially identical to $\mathcal{LOOM}$'s rule, only that we insist that `J` be related to `I` in the declared interface hierarchy.

For two class types, we simply consider the implicit interface type of that class (an interface that contains the signatures of all of its public methods) and consider matching

```
public class Node<T> {
  protected T data;
  protected ThisType next;

  public Node(T data, ThisType next) {
    this.data = data;
    this.next = next;
  }
  public T getData() { return data; }
  public ThisType getNext() { return next; }
  public void setNext(ThisType newNext) { next = newNext; }
}

public class DblNode<T> extends Node<T> {
  protected ThisType prev;

  public DblNode(T data, ThisType next, ThisType prev) {
    super(data, next);
    this.prev = prev;
  }
  public void setNext(ThisType newNext) {
    next = newNext;
    if (newNext != null) newNext.prev = this;
  }
  public void setPrev(TypeType newPrev) {
    prev = newPrev;
    if (newPrev != null) newPrev.next = this;
  }
  public ThisType getPrev() { return prev; }
}
```

Figure 2.9: Linked-list nodes using `ThisType`

```
public class List<T, N extends Node<T>> {
  protected N head;

  public List() { head = null; }
  public void add(T data) {
    head = new Node<T>(data, head);   //actually illegal!
  }
  //pre: list is not empty
  public T remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    N oldHead = head;
    head = head.getNext();
    oldHead.setNext(null);
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 2.10: Generic `List<T, N>`

based on those implicit interface types. Following from this rule, a class type can only match another class type that is higher up on the inheritance tree. A class type `C` matches an interface type `I` if the implicit interface type of `C` matches `I`. Finally, interface types never match class types.

Quite simply, matching tells us which methods can be called on an object with a particular type. Thus, it is very similar in most cases to the regular Java rules for subtype polymorphism.[3] In fact, for classes that do not contain any `ThisType` expressions, matching corresponds exactly to the normal Java rules for subtyping. Thus, we can almost always consider matching as equivalent to the relationship defined by the `extends` keyword in Java. We'll see in the next section how to resolve the problem in the `breakit` example above.

[3] Java does not truly support subtype polymorphism because of problems with the conditional operator. In Java, in an expression of the form `b ? E1 : E2`, `b` must have type `boolean`, and `E1` and `E2` must be compatible types (*i.e.*, one of `E1` or `E2` extends the other, possibly indirectly). This is problematic because if `E1` has type `T extends U` and `E2` has type `S extends U` then `b ? E1 : E2` is not typeable even though it would be if one of `E1` or `E2` had static type `U`. Therefore, a well-typed expression that is written with a type `U` may not always be well-typed if we replace it with an object of type `T`, a subtype of `U`. However, Java does support a limited form of subtype polymorphism for actual parameters and assignments to variables.

2.5 Exact Types

We still have not directly addressed the problem illustrated previously in `breakit`. This hole can be fixed by requiring that the type of the receiver of a binary method must be known exactly. If we know the type of the object exactly, then we can statically determine if the method call is safe – that is, if the parameter is of the same type as the object that the method is invoked on.

Thus, we introduce exact types. If a type expression is declared exactly, then it may only be the receiver of values whose type is identical to it. An exact type is declared by prefacing the type expression with the `@` character. If a method has an argument of type `@C`, then the formal parameter to that argument must have type exactly `C` – not a subclass of `C`. Thus, exact types can be used to limit subtype polymorphism.

Our problem illustrated previously in `breakit` is easily fixed by making the types of both parameters exact types (strictly speaking the rule only requires that the first parameter be exact, as it is the receiver of the binary method, though in practice, both parameters would be exact because `setNext` takes a parameter of type `@ThisType`).

```
void noBreakit(@Node<T> list1, @Node<T> list2) {
  list1.setTail(list2);
}
```

Exact types are clearly needed to safely statically type-check classes that use `ThisType`. Thus, that is our main motivation for adding them to the type system. However another motivation is that they increase the language's expressiveness by explicitly supporting homogeneous data structures. A homogeneous data structure is one where all of the elements are of the same type. For example, to produce a homogeneous list, we declare the element types of our list to be `@T` rather than `T`. A homogeneous list is shown in Figure 2.12 with nodes in Figure 2.11. The problem with constructing a new `head` node in `HomList<T, N>`'s `add` method is discussed in Section 5.2.

```
public class HomNode<T> {
  protected @T data;
  protected @ThisType next;

  public HomNode(@T data, @ThisType next) {
    this.data = data;
    this.next = next;
  }
  public @T getData() { return data; }
  public @ThisType getNext() { return next; }
  public void setNext(@ThisType newNext) { next = newNext; }
}

public class HomDblNode<T> extends Node<T> {
  protected @ThisType prev;
  public HomDblNode(T data, @ThisType next, @ThisType prev) {
    super(data, next);
    this.prev = prev;
  }
  void setNext(@ThisType newNext) {
    next = newNext;
    if (newNext != null) newNext.prev = this;
  }
  void setPrev(@ThisType newPrev) {
    prev = newPrev;
    if (newPrev != null) newPrev.next = this;
  }
  @ThisType getPrev() { return prev; }
}
```

Figure 2.11: Homogeneous list nodes

```
class HomList<T, N extends HomNode<T>> {
  protected @N head;            //N is exact because List<T, N>
                                //is homogeneous in the node types
                                //(and N is indirectly the receiver
                                //of binary method calls)
  public HomList() { head = null; }
  public void add(T data) {
    head = new HomNode<T>(data, head);   //actually illegal!
  }
  //pre: list is not empty
  public @T remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    @N oldHead = head;
    head = head.getNext();
    oldHead.setNext(null);
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 2.12: Homogeneous `List<T,N>`

Chapter 3

Other Extensions of Java

Extending a language to support additional features is not an easy task. In this chapter, we present a summary of related work to extend Java with parametric polymorphism. The projects that are the most closely related to `Rupiah` include Pizza [ORW97], GJ [BOSW98], and NextGen [CS98]. They all share the common goal of adding F-bounded [CCH+89] parametric polymorphism to Java while retaining compatibility with the Java platform. They achieve this goal with varying success and by quite different translation strategies.

3.1 Pizza and GJ: Exploring Translations to Bytecode

Pizza and GJ are two closely related languages that support parametric polymorphism by translation to Java bytecode. Pizza was implemented first, and supports higher order functions and algebraic data types in addition to parametric polymorphism. However, Pizza's main contribution is the exploration of two different methods of translating parametric polymorphism to Java. Both of these translation techniques, a heterogeneous and homogeneous translation, are implemented in the Pizza compiler. Pizza's successor, GJ, takes the most useful feature from Pizza – parametric polymorphism – and supports it in a language more closely related to Java. GJ uses the homogeneous translation exclusively to convert GJ-level source programs to bytecode. In addition to parametric polymorphism, GJ puts a strong emphasis on compatibility with existing Java libraries by providing a simple migration method for using existing Java classes seamlessly with GJ.

3.1.1 Heterogeneous Translation

The first technique introduced in Pizza, the *heterogeneous translation*, translates polymorphism to standard Java by producing a different compiled class for each instantiation of a parametric class. An analysis of the program source code determines which instantiations of a class are needed, and synthesizes a separate class for each one. For example, if a `List<T>` is defined as in Figure 2.5, and the only instantiations of that class are:

```
List<Integer>
List<String>
```

then the heterogenous translation that is implemented in Pizza compiles classes named `List_$String_$` and `List_$Integer_$` and replaces all polymorphic types with these translated monomorphic ones. These classes are structurally identical to the `StringList` and `IntegerList` classes defined in Figures 2.1 and 2.2. This translation is simple to implement as we just replace a parameterized type with the synthesized monomorphic class representing that specific instantiation. Further, because each instantiation of a class is represented by a different class, the full type information of each object is implicitly carried along with it. Therefore, operations which require run-time type information including casts, `instanceof`, and array creation are able to execute correctly in the heterogeneous translation. However, several problems exist with the translation. First, the space requirements are significant because, for each source class, multiple binary class files are produced. Secondly, because each new instantiation may generate new class files, we must have access to the source code for a parameterized class in order to use new instantiations of it.[1] Thus a pure heterogeneous translation may not be a good option for authors of library code, as they must provide source code for their implementation in order to allow each possible instantiation to be translated.

3.1.2 Homogeneous Translation

The second translation introduced in Pizza, and used in GJ, is the *homogeneous translation*. This technique only generates one compiled class per source level class for any number of distinct instantiations of the parametric class. The basic strategy is to translate polymorphic classes into a single generic representation of the class. Then, whenever different instantiations of the class are used, the appropriate type conversions are added as needed. The translation of the `List<T>` class produces a class structurally identical to the code in Figure 2.3 – the `List` class with element type `Object`. Additionally, the compiler inserts

[1] Depending on the implementation of a heterogeneous translation, it may be possible to generate the heterogeneous classes without having access to the source file – that is, if extra type information is stored in compiled classes. However, the traditional implementation of a heterogeneous translation does not support this feature.

//Pizza/GJ Type	//Erased Type
C<T extends Bound, U>	C
T	Bound
U	Object

Figure 3.1: Erasure examples

casts whenever instantiations of parameterized classes are used, as in Figure 2.4. These casts overcome the discrepancy between types in the generic representation of a polymorphic class and types in specific instantiations. Although casts are used in the translation, type safety can be ensured statically as the concept of parametric polymorphism is supported directly in the type system. In fact, GJ provides a "cast-iron" guarantee that any cast inserted by the compiler will always succeed – it is only inserted to satisfy the JVM verifier.

The homogeneous translation uses a technique called *erasure* to replace parameterized types with more general types. A parameterized type is replaced with its base type, where the base type of `C<T, U>` is simply `C`. Inside a class with type parameters, a type expression that is a type variable is replaced with the bound of that variable. Examples of type erasure are given in Figure 3.1. For example, `T` is erased to `Bound`. An unbounded type variable such as `U` in the same figure is replaced with its implicit bound, `Object`. Thus the code that is obtained by compiling a parameterized class using the homogeneous translation is identical to what a programmer would write if hand-coding the list in standard Java.

Translating types to their erased versions is easy; the more complicated part of the process is inserting the appropriate casts to convert erased types to their more specific instantiation types. For example, if the return type of a method is `U`, an unbounded type variable, then its erased type is `Object`. Any call to that method in GJ expects the return type of the method to have the type of `U`'s instantiation, not `Object`. The compiler must insert this cast in order to convert the erased type `Object` to the type of the instantiation. Because the compiler maintains information about instantiations of parameterized classes internally, it can statically determine where casts are needed.

Additional Typing Rules

Two additional restrictions are made in GJ's type system.

First, class type variables may not be used in any static class members.[2] GJ takes the position that it is illegal to use type parameters in static methods because the method is not tied to a particular instance of a class. Thus, the instantiations of type parameters for a static member are undefined. Similarly, a static instance variable is not tied to any

[2] A member refers to a method, instance variable, or inner class.

instance of a class so declaring static variables whose type involves a type variable such as `List<T>` is not allowed.

This rule differs from the heterogeneous translation which supports per instantiation static instance variables. The heterogeneous translation generates a different compiled class file for each instantiation of a parameterized class. In this translation, static instance variables are represented by a different static instance variable in each instantiation. As a result, a static member of a class that has a parameterized type is not shared across all instantiations of the base class, but only among instantiations with identical types. Note that if the class has more than one type parameter, a static member which only involves a single type parameter is not shared across all of the instantiations of that single variable with the same type. It is shared only among objects with identical instantiations of all of the type parameters. For example in the following class:

```
class C<T,U> {
  static List<T> myList;
}
...
C<Integer, String> c1;
C<Integer, String> c2;
C<Integer, Integer> c3;
```

`myList` is shared between `c1` and `c2` but not `c1` and `c3`. It is not clear which approach to parameterized types in static members is most consistent with the Java language.

Secondly, a class may not simultaneously implement a parameterized interface with different types for the same type variable. For example, suppose a class indirectly implements both `I<String>` and `I<Integer>`. If `I<T>` has a method with return type `T`, allowing multiple interface instantiations introduces method binding difficulties. In order to implement both instantiations of the interface, the translated class has to define two methods with identical signatures except for the return types. However, Java does not allow methods to differ only in their return types (though this restriction is not strictly necessary for a safe static type system). It is difficult to both avoid these method binding problems and retain close compatibility with standard Java. Therefore, it is illegal for a class to simultaneously implement an interface with different instantiations of the same type parameter.

3.1.3 Bridge Methods

The simple homogeneous translation described previously is not correct for all parameterized classes. Method binding complications arise when a subclass changes the bound on a type variable or instantiates a superclass's parameter with a fixed type. To illustrate these problems, consider a subclass of `List<T, N>` (as defined in Figures 2.11 and 2.12) that

```
public interface Comparable<T> {
  public int compareTo(T other);
}

public class OrderedList<T extends Comparable, N extends Node<T>>
     extends List<T, N> {
  public void add(T data) {
    N finger = head;
    N prev = null;
    if (head == null) {
       head = new Node<T> (data, head); //actually illegal!
    } else {
      while ((finger != null) &&
             && (finger.getData().compareTo(data) < 0)) {
        prev = finger;
        finger = finger.getNext();
      }
      prev.setNext(new Node<T> (data, finger)); //actually illegal!
    }
  }
}
```

Figure 3.2: `OrderedList<T, N>` definition

implements an ordered list, shown in 3.2. `OrderedList<T, N>` overrides the `add` method so that new elements are inserted into the list so that the list maintains the order specified by `compareTo`. `List<T, N>` includes a method `add` that is translated as:

```
public void add(Object data) { ... }
```

`OrderedList<T, N>` overrides this method with a different body. However, because `T` is bounded by `Comparable<T>`, the type signature for the translation of `add` is different:

```
public void add(Comparable data) { ... }
```

The full translations for `List<T, N>` and `OrderedList<T, N>` are shown in Figures 3.3 and 3.4 respectively. This discrepancy between the source language, where the `add` method is overridden in `OrderedList<T, N>`, and the translation, where `add` is overloaded in `OrderedList<T, N>` causes method binding problems when an `OrderedList<T, N>` is assigned to an object with static type `List<T, N>`.

```
public class List {
  protected Node head;

  public List() { head = null; }
  public void add(Object data) {
    head = new Node(data, head);
  }
  public Object remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    Node oldHead = head;
    head = head.getNext();
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 3.3: GJ translation of `List<T, N>`

Java combines static and dynamic method binding in the following way: the Java compiler determines at compile-time the exact signature of each method it will call. However the selection of the actual method called is deferred until run-time. This implements a feature for instance (*i.e.* non-static) methods known as *dynamic method invocation*. The "dynamic" refers to the fact that this decision about which method to call is deferred until runtime, when the actual type of the object is known. The problems with the homogeneous translation arise when the static type of an expression is less specific than the actual run-time type. Consider the following program fragment:

```
List<Integer, Node<Integer>> ol =
       new OrderedList<Integer, Node<Integer>>(0, null);
Integer i = new Integer(42);
ol.add(i);     //want to call OrderedList's add
```

The statically declared type of `ol` is `List<Integer, Node<Integer>>`. However, we assign an object with type `OrderedList<Integer, Node<Integer>>` to `ol`. Because of Java's method binding, the method expression `ol.add(i)` is bound statically to a method with signature

```
public void add(Object data)
```

because `ol` has type `List<Integer, Node<Integer>>` which only has one method named `add` (with that signature). However, `ol` has run-time type

```
interface Comparable {
  int compareTo (Object other);
}

class OrderedList extends List {
  public void add(Comparable data) {
    Node finger = head;
    Node prev = null;
    if (head == null) {
       head = new Node(data, head);
    } else {
      while ((finger != null) &&
             && (((Comparable)finger.getData()).compareTo(newData) < 0))
      {
        prev = finger;
        finger = finger.getNext();
      }
      prev.setNext(new Node(data, finger));
    }
  }
}
```

Figure 3.4: GJ translation of `OrderedList<T, N>`

```
OrderedList<Integer, Node<Integer>>
```

The method expression `ol.add(i)` should call the overridden method in `OrderedList<T, N>` that inserts the element in order, not simply at the front of the list. However, as the only method with the statically determined signature is the one inherited from `List<Integer, Node<Integer>>`, the incorrect method will be invoked.

This is problematic because at the GJ source level, the method definition for `void add(T data)` is *overridden* in `OrderedList<T, N>` by a method with an identical signature:

```
public void add(T data)
```

The translation should reflect this override. However, the overridden method in the source language becomes an *overloaded* method in the translation. While the program source for `OrderedList<T, N>` declares only one method, with signature `void add(T data)`, the translated version has two.

The solution to this problem is to add a *bridge method* for these problematic methods. In general, three conditions must be met for a method to require a bridge:

- the method overrides a method from a superclass,
- the method signature contains a type variable, or a parameterized type, and
- the bound on the type variable has been changed (tightened) from the declared bound in the superclass or instantiated with a fixed type.

A bridge method overrides the inherited method, and calls the correct definition (the overloaded version) in the subclass. The net effect is that the definition that was translated as an overloaded method definition is always eventually called – operational semantics equivalent to overriding. In the above example GJ inserts a bridge method in `OrderedList<T, N>` that overrides the method inherited from `List<T, N>` with the following definition:

```
public void add(Object data) {
  add(data);            //calls add(Comparable data)
}
```

This solves the method binding issue by ensuring that the method with signature

```
public void add(Comparable data)
```

is always called whenever `add()` is called on an object with run-time type `OrderedList<T, N>`.

The definition of this bridge method appears troublesome because it is not actually legal Java source code. First, we have defined two methods that only differ in their return types, which Java does not normally allow. However, while Java makes this restriction, the

JVM is able to distinguish between methods that differ solely on their return types. GJ's translation exploits this discrepancy between the Java language and the JVM bytecode language, and allow methods to differ only in return types when the compiler synthesizes a bridge method. Thus while the emitted code is not legal Java, it still runs correctly on the JVM.

Second, the call to `add(data)` in the bridge method seems ambiguous, because two methods named `add` are defined for `OrderedList<T, N>`. However, recall that each method is bound to a specific signature at compile time. The GJ compiler never binds a method call to a bridge method. Thus when the compiler binds the call for

```
add(data)
```

it always chooses the method with signature

```
add(Comparable data)
```

because the method with parameter type `Object` is flagged internally as a bridge method. However, the bridge method may be called by dynamic invocation (as in the previously described example that illustrated the problem with `ol.add(i)`). Because the JVM can distinguish between methods that differ only in return types, there is no ambiguity about which method to call.

Bridge methods complete the homogeneous translation by solving method binding complications that can result when type variable bounds change in subclasses.

3.1.4 Retrofitting in GJ

GJ provides a simple mechanism for retrofitting existing Java code with type parameters. The translation of GJ classes is often identical to what a Java programmer would write. Therefore, the compiler provides an option to treat non-GJ classes as if they were polymorphic classes. For example, the standard library class `java.util.Vector` may be treated as a parameterized class `Vector<T>` in GJ by defining the parameterized interface for the class, as partially shown in Figure 3.5. With the information in this interface, GJ can statically type check the use of `Vector<T>` and automatically insert the necessary casts between the pre-existing `Vector` class and user code.

3.1.5 Problems with GJ

GJ has several critical limitations, stemming from its simple homogeneous translation. In general, these problems arise in situations where type information about parametrized types is needed at runtime. As types are erased to their simpler types in compilation, full type information is not available at runtime. In particular, array creation, `instanceof` expressions, and type conversions (casts) do not have the same meaning for GJ types as for Java

```
public class Vector<T>
  implements Cloneable, Serializeable {

  private T[] elementData;  //unchecked

  public void setElementAt (T newValue, int index);
  public T elementAt(int index);
  public boolean contains(T element);
  ...
}
```

Figure 3.5: Retrofitting Vector in GJ

types because of this loss of information. Also, `new` array expressions of type variables are problematic in GJ because the actual type of array that should be created can not be determined until `T` is instantiated. The `instanceof` operator similarly needs detailed type information to check the runtime type of an expression for equality with another object type. Standard casts in Java are partially type checked statically, but an additional runtime check is also performed. All three of these operations cause problems in GJ because information about parameterized types does not exist at runtime.

Array Creation Problems

The translation of `new` array expressions involving type parameters causes problems in GJ. For example, the expression

```
new T[5];
```

can not be translated correctly if `T` is a type parameter. This code fragment should create an array with elements of type `T`. However, in the translation, there is no way to determine the type of `T` dynamically. Thus, the expression can only be translated as:

```
new Object[5];
```

However, this solution is not correct because one can store objects of an incompatible type in the array. Consider the following code snippit:

```
public class C<T> {
  protected Object[] oArray;
  void breakArray() {
    oArray = new T[5];
```

```
    oArray[0] = ``broken'';
  }
}
```

This code should throw an `ArrayStoreException` if `breakArray` is invoked and `T` is instantiated with a type other than `String`. Java allows array types to masquerade as subtypes if their element types are subtypes. This introduces a hole in the static type system and thus requires a run-time check whenever an element is stored in the array. If the element type of the array is incompatible with the object being stored, the exception is thrown. However, because the above snippit would be translated as:

```
public class C {
  protected Object[] oArray;
  void breakArray() {
    oArray = new Object[5];
    oArray[0] = ``broken'';
  }
```

no exception is ever thrown.

Because of this problem, a programmer may not write an expression `new T[...]` in GJ. However, GJ offers two solutions to this problem. First, the programmer may choose to leave the `new` array expression unchecked. This means that the array will be created with the element type of its bound, leading to possible type errors. Alternatively, rather than writing `new T[...]` directly, a "factory" class `gj.lang.reflect.Array` may be used that provides a static method for creating arrays whose element type is a type variable.

Even in the absence of Java's unsafe covariant array subtyping rule, problems still arise in GJ with other operations that require run-time type information. For example, casts and `instanceof` expressions cause problems with array types with generic or parameterized element types (discussed in the next two sections).

`instanceof` Problems

The problem with `instanceof` expressions in GJ is that erasure loses type information. It is impossible for the JVM to check if an object is an instance of a parameterized type using its built-in `instanceof` operator, because the erasure of the type being checked does not contain information about the type parameters. For example, the following two type declarations:

```
anObject1 instanceof List<Integer, Node<Integer>>
anObject2 instanceof List<String, Node<String>>
```

would erase to

```
anObject1 instanceof List
anObject2 instanceof List
```

The translated versions of these `instanceof` tests would have value `true` if `anObject1` and `anObject2` contained any `List`. However, the tests should only be `true` for `List<Integer, Node<Integer>>` and `List<String, Node<String>>` respectively.

Similarly, type parameters to a parameterized class are erased to their bound. The following expression:

```
anObject3 instanceof T
```

where `T` is an unbounded type parameter, would erase to

```
anObject3 instanceof Object
```

The translation of this `instanceof` test would always be `true` because any object type is an instance of `Object`. It should however, be false in all cases except when `anObject3` is really an instance of `T`'s instantiated type.

GJ does not allow `instanceof` expressions involving type parameters because of these problems. Instead, it allows the programmer to perform the checks using a "raw type" – the erased type of a parameterized class. Therefore, it is illegal to write

```
anObject instanceof List<Integer, Node<Integer>>
anObject instanceof T
```

but legal to write

```
anObject instanceof List
anObject instanceof Object
```

if `List` is a parameterized class and `T` is a type variable.

Type Cast Problems

Type cast issues in GJ are almost identical to the problems associated with `instanceof`. The Java Language Specification [GJS96] states that a cast generates a compile-time error if it can be proved that it will never succeed. The JVM also performs a run-time validity check for each explicit cast. A cast that fails the run-time check throws a `ClassCastException`.

The difficulties with casts in GJ occur when a cast expression involves a type variable. Consider the translations of the following casts in a class where `T` is a type parameter:

```
(List<Integer, Node<Integer>>) anObject1;
(List<T, Node<T>>) anObject2;       //T is a type parameter
(T) anObject3;                      //T is a type parameter
```

These expressions would be translated to

```
(List) anObject1;
(List) anObject2;
(Object) anObject3;
```

However, these run-time checks for these translated cast expressions would succeed in many situations where they should throw an exception instead. For example, suppose that these three casts were declared in a class where `T` was instantiated with `Integer`. Also suppose that variables `anObject1` and `anObject2` had type `List<String, Node<String>>`, and `anObject3` had type `String`. Then all three casts should throw exceptions. However, in this translation, all three would incorrectly succeed at runtime. As a result of this problem, as with `instanceof` expressions, GJ does not allow cast expressions that involve type variables because it can not execute them correctly at run-time. However, raw types may be used in cast expressions.

3.2 NextGen: Hybrid Translation

NextGen [CS98] seeks to improve on GJ's implementation of parametric polymorphism by providing support for run-time compatible types. The NextGen translation is based on a hybrid translation that is in part both homogeneous and heterogeneous. For each parameterized class, a homogeneous base class is compiled much in the same manner as GJ. Each instantiation of a polymorphic class also results in two additional generated class files: a wrapper interface and a wrapper class. These wrapper interfaces are used to represent the types of actual instantiations while the wrapper classes are used to create instances of parameterized classes. The constructors for wrapper classes simply forward their arguments to the constructors for the base class.

By storing most compiled code in one homogeneous base class, NextGen limits the amount of redundant, bloated code that is traditionally associated with a pure heterogeneous translation. However, it harnesses the ability of the heterogeneous translation to provide access to run-time type information. The partial translation for a class `Vector<T>`, instantiated once with `Integer`, is given in Figure 3.6.

The wrapper classes defined for each instantiation of a class implicitly carry full type information about parameterized types. Thus, NextGen solves GJ's problems with dynamic type operations. As well as supporting `instanceof` and type cast expressions with parameterized types, NextGen allows expressions such as `new T[]` and `new T()`, where `T` is a type parameter. For example the expression

```
object instanceof Vector<Integer>
```

is trivially translated to an `instanceof` expression that compares `object` with the wrapper interface.

```
//NextGen code
public class Vector<T> {
  private T[] elts;

  public Vector(int initCapacity) { ... }
  public T elemenAt(int index) { ... }
  public void setElementAt(T newValue,
                           int index) { ... }
  ...
}
...
  Vector<Integer> vi = new Vector<Integer>(7);

//translation
public abstract class Vector {
  private Object[] elts;
  public Vector(int initCapacity) { ... }
  public Object elementAt(int index) { ... }
  public void setElementAt(Object newValue,
                           int index) { ... }
  ...
}

public interface $Vector$_Integer_$ {}

public class $$Vector$_Integer_$ extends Vector
  implements $Vector$_Integer_$ {
  public $$Vector$_Integer_$(int initCapacity) {
    super(initCapacity);
  }
  ...
}
...
  $Vector$_Integer_$ vi =
     new $$Vector$_Integer_$(7);
```

Figure 3.6: NextGen translation of `Vector<T>`

```
object instanceof $Vector$_Integer_$
```

Similarly, as the instantiation of `T` is known in each heterogeneous wrapper class, an expression of the form

```
object instanceof T
```

can be translated

```
object instanceof Integer
```

in `$$Vector$_Integer_$` and similarly in other instantiations of `Vector<T>`.

However, NextGen still suffers from many of the inefficiencies of heterogeneous translations. While it avoids much of the code bloat traditionally associated with it, NextGen still produces a separate (though admittedly tiny) compiled class and interface for each instantiation of a parameterized class. Thus, it causes similar problems as the pure heterogeneous translation when used in software libraries. NextGen is also not yet implemented in a compiler, making it more difficult to evaluate fully.

3.3 Other Proposals

In addition to these three techniques, several other projects have explored solutions to the problem of adding parametric polymorphism to Java. PolyJ [MBL97], supports constrained polymorphism. However unlike most forms of bounded parametric polymorphism in Java, PolyJ uses constraints specified by structural properties ("where" clauses). Another project, [AFM97] adds run-time compatible parameterized types to Java, but requires a modified `ClassLoader`. A third proposal [SA98] implements homogeneous translation based on reflection that requires modifications to the JVM. Finally, LM [VN00] improves the run-time efficiency of a purely reflective solution by caching the results of most reflective calls in static instance variables.

3.4 Evaluating Proposals

The number of proposed extensions to Java suggests that key changes to the type system are needed as the language evolves. However, the priorities of each proposal are each slightly different and illustrate the many possible directions that the language could go in the future.

The heterogeneous and homogeneous translations each have advantages and disadvantages. The heterogeneous translation is conceptually simpler, because it is equivalent to macro expansion, and the classes it produces work as well at runtime as standard Java classes. However, it has the disadvantage of producing different classes for each instantiation of a parameterized class. This is wasteful both in space and, more problematically, compilation efficiency.

The homogeneous translation is more complex than the heterogeneous translation, as casts must be inserted when the translated class is used. Bridge methods must also be added to make the translation work correctly with Java's method binding. Type erasure also causes problems in operations that require run-time type information. Despite these limitations, the homogeneous translation is a closer fit with Java's compilation semantics because only one class is compiled for each parameterized source-level class definition. This major advantage, coupled with major space and compilation efficiency benefits, make it our preferred translation technique for `Rupiah`.

One implementation of the homogeneous translation, GJ, holds compatibility and smooth interaction with existing Java code as its highest priority. In some ways, GJ is the language most closely related to Java, which is reflected in the simplicity of its translation. Importantly, GJ only compiles one class per source class. Also, it provides an easy way to retrofit existing Java classes with type parameters. The close relationship afforded to existing pre-compiled code is a major selling point of GJ.

A hybrid approach, NextGen, aims to solve the problems associated with a pure homogeneous translation by providing support for runtime operations on parameterized types. It accomplishes this by utilizing a lightweight heterogeneous translation that makes the runtime types of parameterized classes available at runtime. However, many of the problems with a heterogeneous translation also arise. Space efficiency is less of an issue because the wrapper classes are lightweight. Still, efficiency of compilation is a major concern because many classes must be recompiled whenever a change is made in a class with type parameters. A second concern is that the translation itself is much more complex than either Pizza or GJ. As a result, NextGen's retrofitting mechanism is not as simple as GJ's. Overall NextGen does a good job of optimizing time and space complexity while still providing runtime support for parameterized types at the cost of a very complex language and translation.

These proposals all share the same fundamental goal: adding genericity to Java while retaining maximum compatibility with existing Java code and semantics. They differ however, in the method to reach that goal, and the degree to which they achieve it. Our main complaint about GJ is that the run-time semantics of some operations (casts, `instanceof`, and `new` array expressions) are not correct for all types in the language. NextGen does not have these problems, but suffers from serious space and compilation inefficiencies.

In this thesis, we focus on adding language features to `Rupiah`. Our goal is to make the type system more expressive to allow object-oriented programming in a statically type safe way. Thus we do not limit our focus to parametric polymorphism. These additional features are explored in Chapters 4 and 5.

3.5 Revised Goals

In Section 1.4, we described some preliminary goals for the `Rupiah` project. Having shown some other extensions of Java, we now present more specific goals for the `Rupiah` project.

- **New Expressive Features:** Java's type system has poor support for generic programming and binary methods. Parametric polymorphism and `ThisType` provide type system solutions to these problems. Exact types are also a useful feature, and needed to support `ThisType` in the type system. Finally, as illustrated in Chapter 2, creating objects whose type is a type parameter is difficult. `ThisClass`, described in Chapter 5 provides a solution to this problem. Thus, we propose adding these features to Java.

- **Compatibility with Java:** We chose to implement our language by translating these features to standard bytecode. The features of GJ which allow both backward (existing source files can be compiled with their compiler) and excellent forward compatibility (existing class files can be used, and even retrofitted in GJ programs) are our model for compatibility.

- **Run-time Types, Reflection:** Despite GJ's close compatibility with Java, we believe that any translation of new type system features must satisfy existing semantics for the run-time behavior of type expressions in operations. Thus, `instanceof`, type casts, and `new` array expressions should all operate correctly for new type abstractions. Full reflection for extended types does not seem reasonable for any efficient translation. However, a translation that includes JVM changes could easily support reflection with relatively few changes to the standard reflection classes. In this project however, we do not consider translations that include JVM changes.

- **Reasonable Efficiency:** GJ has the nice property that it is very efficient both in terms of space and run-time execution. Polymorphic classes are compiled to a single representation, thus, there is no code bloat for classes compiled by GJ. There is a small amount of overhead because dynamically checked casts and bridge methods are inserted into class files but these slowdowns are essentially the same as would exist in a hand-coded program. Thus, they are within the acceptable range for execution efficiency. In contrast, the heterogeneous translation has excellent run-time efficiency but it suffers from severe space efficiency problems because a different class (either a complete heterogeneous class, or a lightweight wrapper class in NextGen) is compiled for each instantiation. In this project, we wish to use a homogeneous translation because of its nice space and compilation properties. However, we will balance this with our desire for types that are compatible with introspection operations.

Chapter 4

Introduction to Rupiah

Rupiah [Bur98] is an extension to Java that supports parametric polymorphism via a homogeneous translation to bytecode. It also supports a `ThisType` construct and exact types. Support for parametric polymorphism in `Rupiah` closely follows the implementation technique developed with GJ. However, unlike GJ, type expressions produced by the `Rupiah` compiler are valid in cast and `instanceof` expressions. In this chapter, we describe the implementation of the language's core features including parametric polymorphism, `ThisType`, and exact types.

The implementation described in this chapter is due to Burstein [Bur98]. However, many of the specific translations have been updated.

4.1 Run-time Support For Parameterized Types

The main advantage of `Rupiah`'s translation over GJ's is that new abstracted types can be used in exactly the same way as regular Java types. As shown in Section 3.1 the use of erasure alone in GJ means that operations such as type casts and `instanceof` no longer work as expected because the extended type information is lost at compile time. This problem is fixed in `Rupiah` by making type information about type parameters available to classes at run-time. The strategy with this approach is to store a representation of its full parameterized type with every polymorphic class. Then, we translate each operation which needs run-time type information using the stored representation of parameterized types as needed.

To accomplish this goal, we define the `PolyClass` class, to represent polymorphic types. Just as a `Class` object represents types at run-time in standard Java, a `PolyClass` object

represents a type in Rupiah. However, a type in Rupiah need not be a simple class type. Because Rupiah supports parametric polymorphism, the type may be an instantiation of a parameterized class. Thus, our representation of types in Rupiah must be able to represent these more complicated types. The important elements of the PolyClass object are the base class – for example List is the base type of List<T>, and an array of PolyClass objects that hold the instantiations of type parameters (*i.e.*, when a type variable is itself instantiated with a parameterized type). The partial source code for PolyClass is shown in Figure 4.1.

For each type parameter declared in a class, a corresponding private instance variable of type PolyClass is inserted to store the run-time type of that type parameter. For example, if a class has a type parameter T, then the following declaration:

```
private EDU.williams.rupiah.PolyClass T$$class;
```

is added to the class.

PolyClass objects are initialized in constructors for parameterized classes. For each type parameter of a class, an extra argument of type PolyClass is inserted by the Rupiah compiler into constructors for the class. An assignment is performed in the body of the constructor of arguments to corresponding PolyClass instance variables. This technique is shown in Figure 4.2.

Calls to constructors for parameterized classes are also modified to include PolyClass objects, as illustrated in Figure 4.3. These objects may be already existing PolyClass objects, or objects whose initialization statement is synthesized by the compiler. As Figure 4.3 illustrates, constructing a parameterized class outside of that class's definition results in the creation of a new PolyClass object. In a parameterized class, the existing type parameters of that class may be reused in the creation of other parameterized classes. For example, in the new List<T>() expression, the PolyClass object T$$class representing T's run-time type is reused and passed to the constructor for List. Because any instantiation of List already has a PolyClass representing T, no new PolyClass object need be constructed. Inside constructors, the compiler also passes PolyClass objects to super() or this() expressions. This ensures that the private PolyClass objects of polymorphic superclasses are properly initialized.

4.1.1 instanceof Expressions

With the addition of information about type variables to Rupiah classes, we can now distinguish between two different instantiations of a class such as List<Integer> and List<String> at run-time. Thus, the instanceof operator can be translated to have the correct behavior. Rupiah extends the instanceof operator for parameterized types into a two-step process.

```
public final 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 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 4.1: Java definition of `PolyClass`

```
/* Rupiah Version */
public class List<T> {

  protected Node<T> head;

  public List() {

    head = null;
  }
  ...
}
```

```
/* Translated Version */
public class List {
  private PolyClass T$$class;
  protected Node head;

  public List(PolyClass T$$class$) {
    T$$class = T$$class$;
    head = null;
  }
  ...
}
```

Figure 4.2: Adding type parameter arguments to constructors

```
/* Rupiah Versions */
List<Integer> li = new List<Integer>();
List<String> ls = new List<String>();
List<List<String>> lls = new List<List<String>>();
List<T> ltn = new List<T>();

/* Translated Versions */
List li = new List(new PolyClass(Integer.class));
List ls = new List(new PolyClass(String.class))
List lls = new List(new PolyClass(List.class,
            new PolyClass[] { new PolyClass(String.class) }));
List ltn = new List(T$$class);
```

Figure 4.3: Initializing `PolyClass` objects for constructor calls to `List<T>`

The `instanceof` operator for parameterized classes has the following semantics. An object of type `C<T>` is an `instanceof D<U>` if `C` is a subtype of `D`, and `C`'s instantiation of `D`'s parameter is `U`. While instantiations of type variables must be invariant, a class need not actually have the same type variables as another class in order to be an extension of that class. Consider this example:

```
class C<T> { ... }
class D extends C<String> { ... }
```

Even though `D` does not have a type variable, `D` is an extension of `C<String>` because it instantiates `C`'s type parameter `T` with fixed type `String`. To evaluate an expression of the form

```
anObject instanceof List<Integer>
```

We first check that `anObject` is an `instanceof` the base type, `List`. Then we check that `anObject`'s instantiation of `List`'s type parameter is `Integer`.

The check of the base type is easy to do using the existing `instanceof` operator. The erasure of a parameterized type is its base type. Thus, any object that was originally a `List<T>` will be a `List` in the translation. However, checking the run-time instantiations of type parameters is more difficult. To aid these checks, the compiler adds a synthesized method for each parameterized class. For a class (or interface) `C`, a method named `$instanceOf$C` method is added. This method takes the same number of `PolyClass` arguments as type parameters to that class. The body of this method checks whether the passed arguments are equal to the instance variables representing the instantiated type parameters. For example, the `$instanceOf$List(PolyClass other)` method of `List<T>` checks if the passed `PolyClass` is equal to the `PolyClass` representing `T` – `T$$class`. Additionally, a class defines `$instanceOf$I` methods for any interfaces that it implements. Examples of synthesized `$instanceOf$` methods are shown in Figure 4.4 for two class declarations. Note that an `$instanceOf$C` method is not required in `D` because it is inherited from `C` while `$instanceOf$I` must be defined because `I` is an interface and thus can not itself define the body of the method.

With these synthesized methods we can check the run-time values of instantiations of type parameters in a class. The `instanceof` operator for parameterized types becomes:

- check that the object is an instance of the base type, and
- check instantiations of type parameters by casting the object to the base type, and invoking the appropriate `$instanceOf$` method.

Some examples of the full translation of `instanceof` for parameterized types are shown in Figure 4.6. Note that checking if an object is an instance of a type parameter requires a purely reflective solution because we do not statically know the base type of an instantiation of a type parameter. The method with signature

```
/* Rupiah Version */
public class D<X, Y> extends C<X> implements I<Integer> { ... }

/* Translated version */
public 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 $instanceOf$I(PolyClass a) {
    return a.getBase() == Integer.class;
  }
}
```

Figure 4.4: Adding `$instanceOf$` methods to classes

```
boolean $instanceOf$(Object other)
```

in `PolyClass` (shown in Figure 4.5) performs the reflective call and returns true if `other` is an `instanceof` the instantiation of the type parameter at run-time.

Side Effects

The `instanceof` operator in `Rupiah` is translated as a two-step process, performing the checks described above. Thus, the expression being checked must be evaluated multiple times. If the expression represented by `object` is an identifier expression, then this is no problem. However, if the expression has side effects, then this translation will evaluate the expression (and the side effects) multiple times. To compensate for this possible problem, we introduce a synthetic variable into the translation. This variable is assigned the value of the expression in the first step of the translation of `instanceof`. Then the synthetic variable is used in the additional portions of the translation so that the actual expression being checked is only evaluated once.

For example, in Figure 4.6, the test for

```
if (object instanceof List<String>) { ... }
```

translates to

```
Object $synth_0$;
if ((($synth_0$ = object) == null)
```

```
public boolean $instanceOf$(Object other) {
  //check base type
  if (!base.isInstance(other)) return false;

  //check param types
  if(params.length == 0) { return true; }
  Class[] methTypes = new Class[params.length];
  for (int i=0; i < params.length; i++)
    methTypes[i] = PolyClass.class;
    try {
      Method m = base.getMethod(``$instanceOf$'' + getShortName(base),
                                methTypes);
      Boolean b = (Boolean) m.invoke(other, params);
      return b.booleanValue();
    } catch (Exception e) { return false; }
}

public String getShortName(Class c) {
  //returns the unqualified name (without package information) of c
}
```

Figure 4.5: `PolyClass.$instanceOf$` method

```
// Rupiah versions
if (object1 instanceof List<String>) { ... }
if (object2 instanceof List<T>) { ... } //T is a type parameter
if (object3 instanceof T) { ... }       //T is a type parameter

// Translated versions
Object $synth_1$;
if ((($synth_1$ = object1) == null)
  || (($synth_1$ instanceof List)
  && ((List)$synth_1$).$instanceOf$List(new PolyClass(String.class)))) { ... }

Object $synth_2$;
if ((($synth_2$ = object2) == null)
  || (($synth_2$ instanceof List)
  && ((List)$synth_2$).$instanceOf$List(T$$class))) { ... }

T$$class.$instanceOf$(object3) { ... }
```

Figure 4.6: `instanceof` translation for polymorphic types

```
    || (($synth_0$ instanceof List)
      && ((List)$synth_0$).$instanceOf$List(new PolyClass(String.class))))
```

Full translations for other `instanceof` expressions are shown in Figure 4.6.

4.1.2 Type Conversion Expressions

`Rupiah` also supports type casts that contain parameterized types. Recall from Section 3.1.5 that a cast that fails at run-time should throw a `ClassCastException`. GJ would have problems throwing these exceptions for cast expressions involving type parameters. For instance in GJ one would expect that the cast expression

```
(List<Integer>) anObject;
```

would erase to

```
(List) anObject;
```

and thus would succeed (improperly) if `anObject` is any instantiation of `List`. As a result of this problem, GJ only allows cast expressions involving "raw types" – that is, `List` not `List<Integer>`.

```
//checked cast expression

((C)anObject)


//semantically equivalent expression
//in a language without checked casts
//(ie (C)anObject is not dynamically checked)

Object $synth_0$;
  ...

((($synth_0$ = anObject) instanceof C)
   ? (C)$synth_0$ : throw new ClassCastException())
```

Figure 4.7: Relationship between `instanceof` and explicit cast expressions

`Rupiah` properly supports checked type casts by using a similar technique as is used to support the `instanceof` operator. In general, if a language has an `instanceof` operator, then dynamically checked casts can be simulated using the code shown in Figure 4.7. `Rupiah` is able to determine the value of `instanceof` expressions involving type parameters. Thus the extended `instanceof` operator can be used to execute dynamically checked casts. Examples of the translation of checked casts in `Rupiah` are given in Figure 4.8.

This technique completes the type cast only after the run-time validity check has been performed, using the extended `instanceof` operator. As with `instanceof` expressions, synthetic variables are inserted as needed so that casts of expressions with side effects are only evaluated once.

4.2 Exact Types

`Rupiah` supports *exact types*. Exact types allow the programmer to specify types more precisely as well as being a necessary addition to support the translation of `ThisType`, described in Section 4.3. Exact types are translated to unexact types; that is, `@T` erases to `T`. The type-checking for exact types is done at compile time. For example, if a method parameter is declared with type `@T`, then the compiler only allows that method to be invoked with an actual parameter that is also declared to have type `@T`. However, operations such as `instanceof` and casts which make use of run-time type information require more complicated run-time checks.

```
//Rupiah versions
List<Integer> li = (List<Integer>) object1;
List<T> lt = (List<T>) object2; // T is a type parameter
T t = (T)object3;               // T is a type parameter
                                // with bound B

// Translated versions
Object $synth_1$;
List li = ((($synth_1$ = object1) == null)
           ||
          ((List)$synth_1$).$instanceOf$List(new PolyClass(Integer.class)))
            ? (List)$synth_1$
            : throw new ClassCastException();

Object $synth_2$;
List lt = ((($synth_2$ = object2) == null)
           ||
          ((List)$synth_2$).$instanceOf$List(T$$class))
            ? (List)$synth_2$
            : throw new ClassCastException();

Object $synth_3$;
B t = (T$$class.$instanceOf$($synth_3$ = object3))
         ? (B)$synth_3$
         : throw new ClassCastException();
```

Figure 4.8: Validity checks for casts in Rupiah

```
//Rupiah
anObject1 instanceof @C
anObject2 instanceof @D<T>
anObject3 instanceof @T

//Translation
Object $synth_1$;
((($synth_1$ = anObject1) == null)
  || ($synth_1$.getClass() == C.class))

Object $synth_2$;
((($synth_2$ = anObject2) == null)
 || (($synth_2$.getClass() == D.class)
      && (((D)$synth_2$).$instanceOf$D(T$$class)))

T$$class.$instanceOf$(anObject3, true)
```

Figure 4.9: `instanceof` translations for exact class types

4.2.1 `instanceof` with Exact Types

`Rupiah` correctly supports the `instanceof` operator for expressions involving exact types. An object is an instance of an exact class type if it really is an instantiation of that class. The JVM only creates one `Class` object to represent each class at run-time. Thus, if the right side of an `instanceof` expression is an exact class type, then the `Class` object representing the expression being checked can be compared with the `Class` object representing the right side's type. For example, the translation of `object instanceof @C` is:

```
object.getClass() == C.class
```

If the target is a parameterized class type, for example `C<T>`, then we must also check the instantiations of type parameters. We use the `$instanceof$C` methods, introduced in Section 4.1.1, to check instantiations. Examples of these translations are shown in Figure 4.9.

If the type being checked is an exact parameter type, then a reflective solution is used. We modify the `$instanceOf$` method of `PolyClass` so that it can handle exact types properly. A `boolean` instance variable indicates if the check should be performed exactly or not. This revised method is shown in Figure 4.10.

If the target of an `instanceof` expression is an exact interface type, then a more complex translation is needed. The run-time type of an object in Java is never an interface type

```
public boolean $instanceOf$(Object other, boolean exact) {
  //check base type
  if (exact) {
    if (!(base == other.getClass())) return false;
  } else {
    if (!base.isInstance(other)) return false;
  }

  //check param types
  if(params.length == 0) { return true; }
  Class[] methTypes = new Class[params.length];
  for (int i=0; i < params.length; i++)
    methTypes[i] = PolyClass.class;
    try {
      Method m = base.getMethod(``$instanceOf$'' + getShortName(base),
                                methTypes);
      Boolean b = (Boolean) m.invoke(other, params);
      return b.booleanValue();
    } catch (Exception e) { return false; }
}
```

Figure 4.10: Revised `PolyClass.$instanceOf$` method for exact class types

```
static 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 4.11: `PolyClass.hasExImp` method

(as an interface can not be constructed) so we can not simply compare `Class` objects for equality at run-time. For exact interfaces, we introduce the idea of an *exact interface implementation*. A class in `Rupiah` may implement an interface exactly, which means that the public methods of the class are identical to the interface's public members. A class may only implement one interface exactly. If it were allowed to implement multiple exact interfaces, the interfaces would be structurally identical, only differing in name. Because Java does not support structural type relationships, we are forced to impose this limit on exact interface implementation to avoid complications in our translation.

Using the concept of exact interface implementation, `instanceof` operations involving exact types can be performed correctly. To check if an `object` is an instance of an exact interface `@I`, the following two conditions must hold:

- `object` has an exact interface implementation.
- `object`'s exact interface is `I`

In order to check the first condition, Java's reflection API is used. The `Rupiah` compiler inserts a `private boolean` instance variable named `$$ExImp` for each class that implements an exact interface. To check that a class has an exact interface the compiler synthesizes an expression that checks if a variable with this name is declared for the class. The method `Class.getDeclaredFields()` is used to obtain an array of `java.lang.reflect.Field` objects. If the class implements an exact interface, then this array contains the `$$ExImp` instance variable. The static method `PolyClass.hasExImp`, shown in Figure 4.11, checks whether an array of `Field` objects contains an element named `$$ExImp`.

Additionally, the exact interface is placed in the first element of the `interfaces` array for that object (ahead of other non-exact interfaces that the class might implement). Thus if a class implements an interface exactly, it will be the first element in the expression `getInterfaces()`, which returns an array of `Class` objects. To check the second condition,

we compare the `Class` object in the first slot of the `interfaces` array with the `Class` object of the interface being checked, shown in Figure 4.13. If the interface is parameterized, then the same checks from Section 4.1.1 are used to check the instantiations of type parameters.

Finally, we must reconsider (for the last time) the translation for

```
object instanceof @T
```

where `T` is a type parameter. Recall that `instanceof` expressions are performed using reflection because `T`'s instantiation type is not known statically. Thus, we must extend our reflective solution to be able to handle exact interface types. As before, this solution catches any exceptions that the reflective calls throw internally and returns `false` for the `instanceof` expression. This is correct because any exceptions that might be thrown by evaluating the object are raised before the method is invoked. We revise the `$instanceOf$` method of `PolyClass` as shown in Figure 4.12.

4.2.2 Type Conversion Expressions with Exact Types

Support for type conversion expressions involving exact types utilizes the translation for `instanceof`. We have already seen, in Figure 4.7, how checked casts can be simulated with a correct `instanceof` operator. Thus, to support a cast to an exact type, we simply perform the same tests that we perform for `instanceof`, if the tests return false, we throw a `ClassCastException`. Examples of the translations for casts to exact types are shown in Figure 4.14.

4.3 ThisType

`Rupiah` also adds a keyword, `ThisType` that stands for the interface type of `this` (`ThisType` is described in Chapter 2). `ThisType` is based on a construct borrowed from $\mathcal{LOOM}$. The strategy used to translate `ThisType` is similar to that used for parametric polymorphism, making use of type erasure, synthesized casts and bridge methods.

4.3.1 Type Erasure and Exact Interface Types

Unlike the translations for parametric polymorphism, the erasure for `ThisType` is not immediately clear. Recall that `ThisType` stands for the interface type of `this`. Importantly, it is not the type of the class where it is declared, but rather the public interface of that type.

In `Rupiah`, in order to ensure that a representation for `ThisType` exists, the compiler requires that every class that uses `ThisType` exactly implement an interface. The programmer may specify an exact interface, or the compiler can synthesize one for them. For a class `C`, the synthesized exact interface, if needed, has the name `$$CIfc`. The synthesized

```
public boolean $instanceOf$(Object other, boolean exact) {
  boolean baseOkay = false;
  //check base types
  if (exact && base.isInterface()) {
     baseOkay = hasExImp(other.getDeclaredFields())
                && other.getClass().interfaces()[0] == base;
  }
  else if (exact) { baseOkay = (other.getClass() == base); }
  else { baseOkay = base.isInstance(other); }

  if (!baseOkay) return false;

  //check param types
  if(params.length == 0) { return true; }
  Class[] methTypes = new Class[params.length];
  for (int i=0; i < params.length; i++) methTypes[i] = PolyClass.class;
  try {
       Method m = base.getMethod(``$instanceOf$'' + getShortName(base),
                                 methTypes);
       Boolean b = (Boolean) m.invoke(other, params);
       return b.booleanValue();
  } catch (Exception e) { return false; }
}
```

Figure 4.12: Final revision of `PolyClass $instanceOf$` method for exact interface types

```
/* Rupiah Version */
anObject1 instanceof @I    //I is an interface
anObject2 instanceof @I<T> //T is a type parameter
anObject3 instanceof @T    //T is a type parameter, possibly instantiated
                           //with an interface type

/* Translated Version */
Object $synth_1$;
((($synth_1$ = anObject1) == null)
|| ((PolyClass.hasExImp($synth_1$.getClass().getDeclaredFields()))
    && ($synth_1$.getClass().getInterfaces()[0] == I.class)))

Object $synth_2$;
((($synth_2$ = anObject2) == null)
|| ((PolyClass.hasExImp($synth_2$.getClass().getDeclaredFields()))
    && ($synth_2$.getClass.getInterfaces()[0] == I.class)
    && (((I)$synth_2$).$instanceOf$(T$$class))))

T$$class.$instanceOf$(anObject3, true);
```

Figure 4.13: Checking `instanceof` with an exact interface

```
/* Rupiah Versions */
(@C) object1            //C is a class
(@C<T>) object2         //T is a type parameter
(@T) object3            //T is a type parameter with bound B
(@I<T>) object4         //I is an interface, T is a type parameter

/* Translated Versions */
Object $synth_1$;
((($synth_1$ = object1) == null)
  || ($synth_1$.getClass() == C.class))
    ? (C)$synth_1$
    : throw new ClassCastException())

Object $synth_2$;
((($synth_2$ = object2) == null)
  || (($synth_2$.getClass() == C.class)
      && ((C)$synth_2$).$instanceOf$C(T$$class))))
  ? (C)$synth_2$
  : throw new ClassCastException()

Object $synth_3$;
T$$class.$instanceOf$($synth_3$ = object3)
  ? (B) $synth_3$
  : throw new ClassCastException()

Object $synth_4$;
((($synth_4$ = object4) == null)
 || ((PolyClass.hasExImp($synth_4$.getDeclaredFields())
     && ($synth_4$.getInterfaces()[0] == I.class)
     && ((I)$synth_4$).$instanceOf$I(T$$class)))
 ? (I) $synth_4$
 : throw new ClassCastException()
```

Figure 4.14: Translation of casts to exact types

interface extends any other interfaces that C implements as well as the exact interfaces of a possible superclass, if it exists. An example using synthesized interfaces is given in Figure 4.15. In this example, the synthesized interface for C only contains C's one public member, cMethod(). However D's synthesized interface is not as simple. Note in particular that $$DIfc extends the exact interface, $$CIfc of D's superclass C. Also, $$DIfc extends each interface that D implements, namely A.

A class's exact interface, either synthesized or explicitly declared, must contain exactly the public, non-constructor members of the class. ThisType expressions are translated to the exact interface of the class. This translation of ThisType to exact interfaces is intuitive because the exact interface by definition represents the public interface of a class and thus is equivalent to ThisType.

One additional rule is required to support this translation of ThisType. The actual type referred to by an expression of type ThisType changes in subclasses. Thus, the erasure of ThisType in a subclass must be related to the erased version of ThisType for its superclass. Therefore, if a class implements an exact interface, that interface must extend the exact interface of its superclass, if it exists.

In general, it is wise not to mix synthesized and declared exact interfaces. If a class D uses ThisType and is declared with exact interface DExIfc, but D's superclass C, which also uses ThisType, does not declare an exact interface then a problem will likely occur. Though C does not declare an exact interface in the source file, the compiler automatically synthesizes an exact interface, $$CIfc. Thus when D is compiled, an error occurs because D's exact interface does not extend the exact interface of its superclass. It is easiest for programmers who choose to use ThisType in Rupiah programs to always use declared interface types, not class types, in type expressions. Many object-oriented languages require that programmers separate interface from implementation in this fashion [Wal96]. However, should programmers wish to use ThisType without considering interface types, the synthetic interfaces are provided. Mixing the two approaches, however, is not recommended.

Like the translations for parametric polymorphism, the simple erasure of extended types is not enough. Often casts are needed to bridge the gap between generalized erased versions of type expressions and more specific types. Consider the code shown in Figure 4.15. In that example, cMethod(), which has return type ThisType, is inherited by D from C. The return type for the method in the translation is $$CIfc – C's exact implementation which represents the type of ThisType in C. As D is a subclass of C, the meaning of ThisType should change for any methods or instance variables inherited from C. Thus a return type $$CIfc for a method inherited from C might require a cast to $$DIfc if it is invoked on an object of type D. The compiler inserts casts as necessary to correctly reflect ThisType's change in type in subclasses.

Still, this translation is not complete. In some cases, bridge methods must be added for methods that contain ThisType in their type signature. Suppose that D overrides cMethod(). The method cMethod(), that was defined in C to return type $$CIfc has return

```
/* Rupiah Version */
public class C {
  ThisType cMethod() { ... }
  ...
}

public class D extends C
  implements A {
  ThisType dMethod() { ... }
  ...
    ThisType tt = cMethod();
  ...
}
```

```
/* Translated Version */
public interface $$CIfc {
  $$CIfc cMethod();
}
public class C implements $$CIfc {
  $$CIfc cMethod() { ... }
  ...
}

public interface $$DIfc
  extends $$CIfc, A {
  $$DIfc dMethod();
}

public class D extends C implements $$DIfc {
  $$DIfc dMethod() { ... }
  ...
    $$DIfc tt = ($$DIfc) cMethod();
  ...
 }
```

Figure 4.15: Synthesizing exact interfaces, adding casts to the translation for `ThisType`

```
public class C {
  ThisType cMethod() {
    ...
  }
}

public class D extends C {
  ThisType cMethod() {
    ...
  }

    C myC = new D();
    ThisType tt = myC.cMethod(); //want to call D's overriden method
                                 //but as myC has static type C, method
                                 //call is bound to C's cMethod()
}
```

Figure 4.16: Method binding problems with `ThisType`

type `$$DIfc` in `D`. This kind of change in a return type, which is normally not allowed in standard Java, is allowed when `ThisType` is used as a return type. Further, because the JVM is less restrictive than the Java language, there is no problem in allowing a covariant change in return types in the translation. Like the translation for parametric polymorphism however, a method binding problem arises because two different method definitions for `cMethod()` exist in the translation of `D` while only one existed in the source class.

As in the example for the `add` method described in Section 3.1.3, `D`'s version of `cMethod` should always be called, even if a `D` is assigned to an object with static type `C`. An illustration of this problem is shown in Figure 4.16. Because Java statically binds the method call to a particular signature, the compiler binds the call `myC.cMethod()` to `C`'s definition.

The problem stems from the fact that `D` overrides `cMethod` but also inherits a definition with a different type signature from `C`. Thus, while `D` had only one `cMethod` defined in the source language, the translation effectively has two methods named `cMethod`:

```
$$CIfc cMethod() { ... } //inherited
$$DIfc cMethod() { ... } //overloaded
```

Only the method with signature

```
$$DIfc cMethod()
```

should ever be invoked on an object with run-time type `D` because `cMethod` is overridden by `D` in the source language.

In order to make sure that the correct method is always invoked, the compiler inserts a bridge method in `D`:

```
$$CIfc cMethod() { return cMethod(); } //calls $$DIfc cMethod()
```

This bridge method ensures that the correct definition of `cMethod()` is called even if method binding is not correct at compile time. As in Section 3.1.3, even though this bridge method is not legal Java code because two methods are defined that differ only in their return types, the JVM can distinguish between the two. Also, as with bridge methods in GJ, the compiler never statically binds a method call to a bridge method. Therefore, the apparent infinite loop in the definition of the bridge method is in fact, not an issue.

4.3.2 Run-Time Support for `ThisType`

In `Rupiah`, `ThisType` expressions evaluate correctly in type casts and `instanceof` expressions. As the type of `ThisType` changes meaning in subclasses, we have to be careful that our translations of `instanceof ThisType` and casts to `(ThisType)` are correct for code that is inherited.

`instanceof` Expressions

To check an `instanceof ThisType` expression correctly, we need to check if the object is an instance of the class's fully instantiated exact interface. However, as `ThisType` changes meaning in subclasses, we do not wish to check a specific exact interface. If a class has exact interface `I`, for example, we can not invoke `$instanceOf$I` directly because the `instanceof` code would not be correct when inherited by a subclass with a different exact interface. Thus we wrap up the exact interface checked in a method and use Java's dynamic method invocation in order to call the method that checks the correct interface. Each class that uses `ThisType` has a method added to the class file with signature:

```
public boolean $instanceOf$ThisType(Object other, boolean exact)
```

This method is overridden by each subclass and checks that `object` is an instance of their exact interface. A sample synthesized `$instanceOf$ThisType` method is shown in Figure 4.17. If the `$instanceOf$ThisType` method is called with `exact` equal to true, then the method performs the check for exact interface types as described in Figure 4.13. Figure 4.18 shows the translation of `instanceof ThisType`.

```
class C<X> implements @I<X, Integer> {
  PolyClass ThisType;
  PolyClass X$$class;

  public boolean $instanceOf$ThisType(Object other, boolean exact) {
    if (other == null) { return true; }

    if (exact) {
      if (!((PolyClass.hasExImp(other.getClass().getDeclaredFields()))
        && (other.getClass.getInterfaces()[0] == I.class)))
          { return false; }
    } else {
      if (! (other instanceof I)) { return false; }
    }
    return ((I)other).$instanceOf$I(X$$class,
                                    new PolyClass(Integer.class));
  }

}
```

Figure 4.17: Synthesized `$instanceOf$ThisType` methods in classes

```
/* Rupiah Version */
class C<T> implements @I<T> {
  ...
  object1 instanceof ThisType
  ...
  object2 instanceof @ThisType
}

/* Translated Version */
class C implements I {
  ...
  public boolean $instanceOf$I(PolyClass t) { ... }
  public boolean $instanceof$ThisType(Object other) { ... }
  ...
  $instanceOf$ThisType(object1, false)
  ...
  $instanceOf$ThisType(object2, true)
}
```

Figure 4.18: Translating `instanceof ThisType` in a class with type parameters

```
 /* Rupiah Version */
class C<T> implements @I<T> {
  ...
  (ThisType) object1
  ...
  (@ThisType) object2
}

/* Translated Version */
class C implements I {
  ...
  public boolean $instanceOf$I(PolyClass t) { ... }
  public boolean $instanceof$ThisType(Object other) { ... }
  ...
  Object $synth_1$;
  ($instanceOf$ThisType(($synth_1$ = object1), false))
    ? (I)$synth_1$
    : throw new ClassCastException()
  ...
  Object $synth_2$;
  ($instanceOf$ThisType(($synth_2$ = object2), true))
    ? (I)$synth_2$
    : throw new ClassCastException()
}
```

Figure 4.19: Casts involving `ThisType`

4.3.3 Type Conversion Expressions

The translation for type casts involving `ThisType` follows the strategy illustrated in Figure 4.7. Namely, the `$instanceof$ThisType` methods are called, and a `ClassCastException` is thrown if the check fails. Examples of the translation of casts to `ThisType` and `@ThisType` are illustrated in Figure 4.19.

4.4 Array Types in `Rupiah`

Recall that `Rupiah` stores `PolyClass` objects with each parameterized class or method. These `PolyClass` objects are used in operations that require full type information at runtime. The `PolyClass` objects are created and passed along to constructors for polymorphic

classes. Arrays are also objects in Java. However, they are not constructed by calling a constructor, but by bytecode instructions. Indeed JVMs have much flexibility in how they implement these instructions. Each array object has the following members:

- a final integer `length` field
- a (shallow) `clone` method
- all of the methods inherited from `Object`

Additionally, the `Class` object for an array type may be obtained by writing the type followed by "`.class`" (used in reflection). Thus an array is an object but is not constructed by any constructor. This presents a problem for the `Rupiah` project because it does not allow our general technique of storing `PolyClass` objects with polymorphic classes by passing them to constructors to be implemented for arrays. Because all array operations are implemented by bytecode instructions, we have no way to store a `PolyClass` object with an array. Thus, we have no way to correctly perform `instanceof`, casts, or `new` array expressions for arrays with element types that are parameterized types, type parameters, or `ThisType`.

A solution to this problem is proposed in Section C. However, this proposal is not yet implemented in the `Rupiah` compiler and has several drawbacks including breaking source compatibility with some Java programs.

4.5 LM: Increasing Homogeneous Translation Efficiency

`Rupiah` solves many of GJ's problems with operations that require run-time type information while staying within the paradigm of the homogeneous translation. However, because it uses reflection to access information about run-time types, the translation is less efficient than either GJ or NextGen. One recent proposal, LM, [VN00] suggests that an approach based solely on reflection can be optimized by resolving many of of the instantiations of parameterized classes at load time. In this section, rather than discuss the specifics of LM's translation, we describe the key concepts as they would work if extended to `Rupiah`.

The JVM only creates one `Class` file for each class used in a program no matter how many instantiations of that class are actually constructed. `Rupiah` uses `PolyClass` objects to represent parameterized types but constructs a new `PolyClass` object every time that a polymorphic class is instantiated. This introduces a significant slowdown because new `PolyClass` objects are created each time that a polymorphic class is instantiated. We would like behavior for `PolyClass` objects that is similar to `Class` objects – namely, that only one `PolyClass` object is ever created to represent a given type. A complicated declaration such as

```
List<List<List<List<List<List<Integer>>>>>>
```

```
public class ListTester
  public static void useLists()
    List<Integer> li = new List<Integer>(new Integer(0), null);
    List<Integer> li2 = new List<Integer>(new Integer(0), null);
    List<String> ls = new List<String>(‘‘’’, null);
    List<List<Integer>> lli = new List<List<Integer>>(li, null);
    List<List<String>> lls = new List<List<String>>(ls, null);
```

Figure 4.20: Complex `List<T>` Usage

illustrates the acute nature of the problem: each of `List<Integer>`, `List<List<Integer>>` etc, are constructed separately each time that the expression is encountered.

Also recall that `PolyClass` objects are often tested for equality in `Rupiah`'s translations for `instanceof` and casts for parameterized types. However, unlike `Class` objects, which can be tested for equality with the inexpensive `==` operator, `PolyClass` must be compared using a more costly structural equality test (the `equals` method).

The optimization introduced in LM is to use a separate class, `PolyClassLoader` to handle all creation and access to `PolyClass` objects. The `PolyClassLoader` contains a static `Hashtable` that stores each distinct `PolyClass` object. Classes obtain their `PolyClass` objects from this loader class, which has `static` methods and thus does not have to be instantiated itself. The `Hashtable` is also `static`, thus, all `PolyClass` objects are shared between all classes that use a `PolyClassLoader`.

The `PolyClassLoader` operates as follows. The first time that a `PolyClass` is needed, the `PolyClassLoader` constructs the object and returns it. Additional requests to the `PolyClassLoader` for the same polymorphic type simply return the already created object stored in the `Hashtable`. This optimization reduces the number of `PolyClass` objects that are created. It also allows `PolyClass` objects obtained through the `PolyClassLoader` to be tested for equality using the `==` operator because only one `PolyClass` is ever created for a given type.

This solution is much improved over the pure reflective solution, but can be further optimized. Consider a class that contains the following code, shown in Figure 4.20, and a polymorphic `List<T>` as defined in Figure 2.5. Introducing a `PolyClassLoader` reduces the number of instantiations of `PolyClass` objects in the translation greatly. However, using the same parameterized type multiple times requires multiple method calls to the `PolyClassLoader`. For example, in the above example, the `PolyClass` representing `List<Integer>` is used three times. Borrowing an optimization used with `Class` objects, we can reduce the number of method calls to the `PolyClass` loader by caching `PolyClass`

objects in static fields. Thus, we declare a field in the class `ListTester`:

```
private static PolyClass List$$Integer$$;
```

and replace all accesses of the `PolyClass` representing `List<Integer>` with the following expression (assuming the `PolyClassLoader` has a method `get` which takes a `String` and returns the `PolyClass` object representing that type):

```
(List$$Integer$$ == null)
   ? (List$$Integer = PolyClassLoader.get(``List<Integer>''))
   : List$$Integer$$
```

This optimization reduces the number of method calls that must be performed, and replaces them with static field references. However, note that if the type contains any type variables, this optimization may not be used. For example, to obtain the `PolyClass` for `List<T>`, we must know `T`'s run-time type. Caching the value of `List<T>` in a static field, would mean that different instantiations of `T` use the same `PolyClass` representing `List<T>` – clearly an error. Thus we only use this last optimization for type expressions that do not contain type variables.

4.5.1 Evaluating LM

LM's main focus is improving the efficiency of a `Rupiah`-like language by reducing the degree of reflection needed to implement the translation. It moves much more of the translation to compile time, so that run-time efficiency is greatly improved. However, it does not add any expressiveness to the type system beyond GJ, NextGen, or `Rupiah` and thus is of less direct interest to this thesis.

Chapter 5

New Language Features

Even with the expressiveness gained by adding the new type system features described in Chapter 4, we have strong motivations for introducing a few additional constructs into the `Rupiah` language. First, the description of `Rupiah` in the previous chapter only defines parametric polymorphism for top-level classes. We propose adding support for inner class and method-level type parameters as well. Second, constructing objects whose type is explicitly `ThisType` or a type variable is almost impossible to do within the bounds of the static type system. `ThisClass` virtual constructors provide a constructor that can safely construct objects of these types. Finally, the `Rupiah` compiler does not support separate compilation of class files. Though this is not related to the expressiveness of the language, it is a practical nuisance. By adding bytecode `attributes` to store full type information for a class's members, we can overcome this limitation.

5.1 Complete Parametric Polymorphism

Earlier versions of the `Rupiah` compiler only supported class-level type parameters. This restriction is easily eliminated by extending our existing implementation to support polymorphic inner classes and methods.

5.1.1 Motivation: Complete Parametric Polymorphism

The concept of parametric polymorphism is fairly general. However in Chapter 4, we have only defined parametric polymorphism for top-level classes and interfaces. Obviously supporting parametric polymorphism for top-level classes but not for inner classes is incon-

sistent. Additionally, the concept can be generalized to allow passing types as parameters to methods as well. A number of languages, including ML, **PolyTOIL**, $\mathcal{LOOM}$, Pizza, and GJ support polymorphic methods. We wish to extend Rupiah so that it supports parametric polymorphism both in inner classes and methods.

Inner Classes

There are strong motivations for adding support for polymorphic inner classes. The first reason is simply that implementing parametric polymorphism for top-level classes, while not doing so for inner classes, is an unnecessary and arbitrary restriction. Polymorphic inner classes are so similar to polymorphic classes that it should be easy to extend our implementation.

A second, perhaps more compelling reason is that several programming techniques depend on their inclusion in the language. For example, in Java programmers often define an `Enumeration` as an inner class. An `Enumeration<T>` provides access to the elements of a data structure through the following interface:

```
public inteface Enumeration<T> {
  public boolean hasMoreElements();
  public T nextElement();
}
```

We would like to be able to define a `ListEnumeration` as shown in Figure 5.1. In this example, we use `List<T>`'s `T` internally in defining `ListEnumeration`. We do not need to declare a separate type parameter for `ListEnumeration` because `T` is already defined for any instantiation of `List<T>`. However, the original `Rupiah` language did not allow type parameters from the enclosing class to be used in an inner class. Thus, `ListEnumeration<T>` would have to be a top-level class. While no functionality is lost by making it a top-level class, a common Java technique is to define an `Enumeration` as an inner class. Thus, it would be unfortunate if the polymorphic version of `Enumeration<T>` could not be defined similarly.

Inner classes should also be able to define their own type parameters. Consider a variation of the `ListEnumeration` class shown above, `ListMapEnumeration<U>`, that maps a function over each element before in the enumerations before returning it. The mapping function, defined by an interface `Mapper`:

```
public interface Mapper<T, U> {
  U func(T t);
}
```

declares a method, `func`, that given an element of type `T` returns an element of type `U`. Thus, in order to define our `ListMapEnumeration<U>`, we need a way to specify the types

```
public class NoSuchElementException extends RuntimeException {}

public class List<T> {
  ...
  public class ListEnumeration implements Enumeration<T> {
    Node<T> finger;
    public ListEnumeration() {
      finger = head;
    }
    public boolean hasMoreElements() {
      return (finger == null);
    }
    public T nextElement() {
      if (!(hasMoreElements())) {
        throw new NoSuchElementException();
      }
      T data = finger.getData();
      finger = finger.next();
      return data;
    }
  }
  public Enumeration<T> elements() {
    return new ListEnumeration();
  }
  ...
}
```

Figure 5.1: `ListEnumeration` definition for `List<T>`

```
public class List<T> {
  ...
  public class ListMapEnumeration<U> implements Enumeration<U> {
    Node<T> finger;
    Mapper<T, U> m;
    public ListMapEnumeration(Mapper<T, U> m) {
      this.m = m;
      finger = head;
    }
    public boolean hasMoreElements() {
      return (finger == null);
    }
    public U nextElement() {
      if (!(hasMoreElements())) {
        throw new NoSuchElementException();
      }
      T data = finger.getData();
      finger = finger.next();
      return m.func(data);
    }
  }
  ...
}
```

Figure 5.2: `ListMapEnumeration<U>` definition for `List<T>`

of `T` and `U`. `T` is already specified by the type parameter of the enclosing class, but `U` can not be declared without support for polymorphic inner classes. With language support for type parameters to inner classes, we can write `ListMapEnumeration<U>` as shown in Figure 5.2.

Extending `Rupiah`'s support for parametric polymorphism to inner classes makes the language more expressive, and is useful in several applications. Moreover, it makes our implementation of parametric polymorphism more consistent by supporting parametric polymorphism for any class or interface, not only for top-level classes and interfaces.

Methods

Our motivations for extending our implementation of parametric polymorphism to support polymorphic methods are similar to those for supporting polymorphic inner classes. First, the concept of parametric polymorphism is a good fit with methods. Thus, our implemen-

tation of parametric polymorphism is more consistent if we allow type parameters to be declared for methods rather than restricting their use only to classes. Second, some important programming techniques can only be programmed in a statically type safe way if polymorphic methods are supported by the language.

For example, by extending `Rupiah` to support polymorphic methods, we can define a simpler `map` method for `List<T>`. The `map` method takes a function from `T` to `U` defined by `Mapper<T, U>`, a `List<T>`, and returns the `List<U>` that is obtained by calling `func` on each element of the original list. This is analogous to ML's `map` function with type signature:

```
fn : ('a -> 'b) -> 'a list -> 'b list
```

A polymorphic `map` method in `Rupiah` is shown in Figure 5.3. Type variables and bounds for a method are declared in `<>` brackets just after the modifiers but before the return type declaration. Thus the method declaration for `map` is:

```
public  <U> List<U> map(Mapper<T, U> m)
```

The syntax for polymorphic method invocation is similar. We specify instantiations of type variables in `<>` brackets before the method name. For example to invoke `map` with a `Mapper<String, Integer>`, we write:

```
List<String> aList = ...
Mapper<String, Integer> m = ...
...
List<Integer> lu = aList.<Integer>map(m);
```

Unlike GJ, `Rupiah` does not support inference of instantiations of method type variables. They must always be explicitly stated.

A second example that illustrates why polymorphic methods are useful involves programming an extensible interpreter for a simple language. The source code for such an interpreter as well as commentary on why type system features such as polymorphic methods are needed for static type safety are given in Appendix B.

5.1.2 Implementation: Complete Parametric Polymorphism

Extending our implementation of parametric polymorphism is quite simple. As we already have an existing framework supporting class level type parameters, relatively few changes need to be made. In fact, most of the changes made to support polymorphic inner classes and methods reside in the parser (a grammar for the `Rupiah` language is given in Appendix A). However, a few substantial changes to the internal compiler representation of type parameters and to type checking rules for method overloading and overriding are needed to support these new features.

```
class List<T> {
  ...
  public <U> List<U> map(Mapper<T, U> m) {
    List<U> listu = new List<U>();
    Node<T> finger = head;
    while (finger != null) {
      U data = m.func(finger.getData());
      listu.add(data);
      finger = finger.getNext();
    }
    return listu;
  }
  ...
}
```

Figure 5.3: A `map` method for `List<T>`

Inner Classes

Our main concern in supporting type parameters for inner classes is the issue of scoping. Scoping rules for type variables follows from Java's scoping rules for variables. A non-static nested class may refer to all of the type variables defined by the enclosing class. If both the enclosing and nested classes define a type variable `T` then inside of the nested class a type expression involving `T` always refers to the nested class's variable.

Early versions of the `Rupiah` compiler stored information about type parameters and bounds during compilation by associating a data structure with each source class. Looking up a type variable simply involved checking the structure stored in the source class. However, because we now allow local type variables in nested members of classes, we redesigned this implementation. Instead, the internal compiler representation of type variables and bounds is stored in an environment. At each nested level, the environment is updated with any newly declared type variables. Looking up a type variable during type-checking involves searching through the chain of nested environments until a match is found. Thus, type variables can be declared at any level – not just for classes. Nested structures can access the type variables defined by enclosing structures. Also, Java's standard static scoping rules for variables apply when a type variable is declared with the same name in both a nested and top-level class.

The basic translation for inner classes is identical to the translation for top-level classes. We translate complicated type expressions to their erased versions and insert casts and bridge methods as necessary. `PolyClass` objects representing the types of instantiations

are declared in polymorphic inner classes, and initialized by constructors exactly as with top-level classes. If a nested class and enclosing class both declare a type variable `T`, then a `PolyClass` object with name `T$$class` is defined for both classes. Java's scoping rules ensure that operations that make use of `T$$class` always access the `PolyClass` object representing the correct `T`.

Methods

As with inner classes, support for polymorphic methods closely follows the technique described in Chapter 4. Scoping rules for type variables are identical to those for variables in standard Java. Inside of a method, local type variables hide class type parameters of the same name. However class type variables are always accessible for non-static methods. For example, the expression `this.T`, if `T` is a class level type parameter, refers to the class's `T`. During the static type checking phase, the compiler steps up through the chain of environments (as described for inner class type parameters) to resolve identifiers to types.

We translate polymorphic methods and classes similarly. For each type variable declared by a class, an argument of type `PolyClass` is added to the method's parameter list for each local type variable. For example, the method definition:

```
public void <S, T>foo() {
  ...
}
```

translates to

```
public void foo(PolyClass S$$class, PolyClass T$$class) {
  ...
}
```

The scope of the type variable is limited to the body of the method, thus, we do not need to store the passed `PolyClass` object in a variable. Having it declared as an argument sufficiently implements the scoping rules for `T` as represented by `T$$class`.

Run-time operations that require full type information are translated exactly as before. Inside the body of a method with a local type variable, the `PolyClass` argument hides the `PolyClass` instance variable representing the class's type parameter of the same name, if it exists. However, if inside of a method a programmer writes a type expression `this.T`, then we reference `this.T$$class` – the class's run-time representation of `T`.

Type Checking Rules We must carefully define overriding and overloading for polymorphic methods. First, our translation is based on erasure. Therefore, any two methods that erase to the same signature must have one method that overrides the other. For instance,

```
public class C {
  void foo<T>(T t) { ... }
}

public class D extends C {
  void foo<T extends Number>(T t) {    //illegal change in bound
    t.compareTo(new Integer(17));
  }
}
...
  C myC = new D();
  myC.foo<Object>(new Object());          //causes error when compareTo
                                          //invoked on the new Object()
```

Figure 5.4: Unsafe covariant bound change

one can not define multiple methods with the same name, parameter types, and number of type variables that differ only in the bounds of some type variables in the same class. These methods would all erase to the same representation, and thus we would have no way to distinguish them in the translation.

If a superclass contains a polymorphic method, then a subclass may override it by providing a new definition for the method body. However, the bounds on the method's type variables must remain invariant. Suppose that the bounds could be narrowed by a subclass. The example in Figure 5.4 illustrates the problem that would result if bounds were allowed to change covariantly. Because `T` is unbounded in `C`'s `foo` method, we can instantiate it with any type. In `D`, `foo` is bounded by `Number`. The problem occurs when we assign an object of type `D` to `myC`. This assignment is legal because `D` matches `C`. If we instantiate the type variable for `myC.foo` with `Object` an error occurs when we invoke `foo` with an argument of type `Object`. As shown in the example an error occurs because `D`'s `foo` expects objects that match `Number`. Because `myC` actually has run-time type `D`, this method will be called. However, because `myC` has static type `C`, the type checker allows `T` to be instantiated with any object type. It would be type-safe to allow contravariant (widening) bounds on type parameters to methods. However, this technique is rarely useful. It is unlikely that a programmer would need to widen the bound on a method level type variable. Thus, `Rupiah` implements the consistent rule that bounds on type variables must remain invariant in subclasses.

`Rupiah` allows method overloading for two methods with the same name that differ only in the number of type variables. Because we add a `PolyClass` object to the parameter list of the method's signature for each local type variable, the translation of the two methods

is different, unlike the situation in GJ.

5.2 ThisClass Constructors

Rupiah increases the expressiveness of Java's static type system with parametric polymorphism, ThisType, and exact types. However, while these new type abstractions can be statically type checked, actually constructing objects of their type is often impossible (in the static type system at least). In this section, we describe a "virtual constructor" for Rupiah that addresses these problems.

5.2.1 Motivation: ThisClass Constructors

Recall from Chapter 2 that constructing objects whose type is a type variable is in general difficult to do in Rupiah. The instantiation of the variable is unknown at compile time so assigning any specific constructor call to an expression whose type is a parameter is illegal. The List<T, N> class, reproduced in Figure 5.5 contains an illegal statement:

```
@N head;
...
  head = new Node<T>(data, head);   //illegal
```

because head has type @N. This assignment fails to statically type check because it causes problems if N is instantiated with a type more specific than Node<T>. In fact, assigning a specific constructor to an object whose type is a type variable is never allowed because we could always instantiate that variable with a more specific type than the type of object created by the constructor that would make the assignment fail. Thus, we have no way to construct objects whose type is a type variable. In our List<T, N> we would like the ability to write an expression like new N(...) that constructs an object of type N at run-time. However, new N(...) can not be translated in any reasonable fashion because Java constructors do not factor into subtyping. Thus, even though Node<T> defines a two-argument constructor, there is no guarantee that any subclass defines the same constructor. While in Rupiah we have a run-time representation of N in the N$$class instance variable, we do not have a way to statically ensure that N's actual instantiation will define a constructor with the given argument types.

To safely write List<T, N> in a language like GJ, we would have to redefine add so that it takes a parameter of type N – the node type – instead of T. This is undesirable because we'd like our list to operate on data, not nodes. There are alternative approaches that would allow us to construct something of type N inside of the add method, but none solve the problem completely. First, we could use reflection on N to create an object of the correct type. One could imagine a small set of Rupiah reflection methods for type

```
public class List<T, N extends Node<T>> {
  protected @N head;

  public List() { head = null; }
  public void add(T data) {
    head = new Node<T>(data, head);   //actually illegal!
  }
  //pre: list is not empty
  public T remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    @N oldHead = head;
    head = head.getNext();
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 5.5: Illegal `List<T, N>`

variables that could be used to create an object of type `N` (by using standard Java reflection on `N$$class.getBase()`). Alternatively, we could clone `head` and then update the fields appropriately. With the addition of `ThisType` to our language, we can write `Cloneable` as follows:

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

Thus, the result of a `clone` method does not need to be cast. However, defining the body of `clone` for any class that implements `Cloneable`, or more generally, for any method that returns `ThisType`, is impossible to do without resorting to casts, as shown below.

Problems constructing objects of type `ThisType` illustrate the broad nature of the problem. As with expressions whose type is a type variable, it is very difficult to create objects whose type is `ThisType`. Because `ThisType` changes meaning in subclasses, we can not assign the result of any specific constructor to an expression of type `ThisType`. The example in Figure 5.6 illustrates why this assignment is illegal. If the assignment statement

```
ThisType tt = new C();
```

is executed by a `C` there are no problems. Because `ThisType` is equivalent to `@I`, and `C` implements `@I`, the assignment of a `new C()` to `tt` is safe. However, when `foo()` is inherited

```
public class C implements @I {
  void foo() {
    ThisType tt = new C(); //okay in C, problems
  }                        //when called by D
}

public class D extends C implements @J {
  void bar() {
    foo();      //assignment of new C() to tt fails
  }
}
```

Figure 5.6: Illegal assignment to `ThisType`

and invoked by `D`, `tt` has type equivalent to `@J`. Thus in `D` the assignment statement is equivalent to

```
@J tt = new C();
```

which is illegal because `ThisType` means `@J` in `D` and `C` is not a subtype of `J`. Thus, a specific constructor call may not be assigned to an object of type `ThisType` in `Rupiah`. We can not define the body of the `clone` method as it is declared in `Cloneable` above without including a cast to type `@ThisType`. This limitation makes many programs, including `List<T, N>` difficult to write safely.

These problems with constructing objects are frustrating because the language provides us with new type abstractions such as parametric polymorphism and `ThisType`, but provides no way to construct objects of those types.

Introducing `ThisClass`

If we could write a method that always returned an object of the same type as itself, then the problems described in the previous section would be solved. For example, suppose that objects could define an inheritable method `create` with return type `@ThisType`. This would effectively implement an "inheritable constructor" for that object. With this functionality, we could safely create objects of type `@T`, a type variable, by invoking the `create` method of the instantiation of a type parameter (given a reasonable mechanism to do so). Similarly, objects of type `ThisType` could be created by calling the `create` method of `this`. Even though `ThisType` changes meaning in subclasses, the inheritable constructor would also change in a parallel fashion (`this` also changes which object it refers to in subclasses so `this.create()` would dynamically call the correct method).

```
public class Node<T> {
  protected T data;
  protected @ThisType next;

  public ThisClass(T data, @ThisType next) {
    this.data = data;
    this.next = next;
  }
  ...
}
```

Figure 5.7: `ThisClass` constructor for `Node<T>`

With this concept mind, we introduce `ThisClass` constructors. The idea of a constructor that always creates an object whose type is the same as `this` is due to Bill Joy. We believe that `Rupiah` is the first extension of Java to support this kind of constructor. `ThisClass` constructors are identical to Java constructors, except that they must be overridden by each subclass. Thus, they provide a guarantee that each subclass has at least the `ThisClass` constructors as its superclass. A `ThisClass` constructor has the behavior that it always creates an object whose type is the exactly the same as the type of `this`. They provide equivalent functionality to an inheritable constructor, and can be used to safely initialize expressions whose type is either a type parameter `T`, or `ThisType`.

`ThisClass` constructors are declared by writing a constructor with name `ThisClass`. A sample `ThisClass` constructor for `Node<T>` is shown in Figure 5.7. Because `Node<T>` defines a `ThisClass` constructor, its subclass `DblNode<T>` must also define a constructor with the same argument types, shown in Figure 5.8. Additionally, `DblNode<T>` defines an additional three-argument constructor that initializes the data and both the `next` and `prev` links. `DblNode<T>`'s two-argument constructor, which overrides the `ThisClass` constructor in `Node<T>`, invokes the three-argument `ThisClass` constructor using the `thisThisClass()` keyword. The keywords `thisThisClass()` and `superThisClass()` are analogous to standard Java keywords `this()` and `super()`, except that they always call `ThisClass` constructors, not standard ones.

With `ThisClass` constructors defined for our list nodes, we can finally write `List<T, N>` safely. As shown in Figure 5.9, constructing a new object of type `N` is implemented by a new instance call:

```
new N.ThisClass(data,head)
```

This expression always invokes the correct constructor whether `N` is instantiated with

```
public class DblNode<T> extends Node<T> {
  protected @ThisType prev;

  public ThisClass(T data, @ThisType next) {
    thisThisClass(data, next, null);
  }

  public ThisClass(T data, @ThisType next, @ThisType prev) {
    superThisClass(data, next);
    this.prev = prev;
  }
  ...
}
```

Figure 5.8: `ThisClass` constructor for `Node<T>`

`Node<T>`, `DblNode<T>`, or any other valid instantiation. Note that as `ThisClass` constructors are invoked by referring to static class types (as opposed to being invoked on instances of classes), we need not worry that `Node<T>`'s `ThisClass` constructor is a binary method. The "receiver" of the binary method in this case is not a particular object but a type variable `N`. `N` is effectively an exact type because it *is* the type of its instantiation and never a subclass. Thus, the compiler allows binary method calls for `ThisClass` constructors.

5.2.2 Implementation: `ThisClass` Constructors

We have two main concerns in implementing `ThisClass` constructors. First, we wish to distinguish between `ThisClass` constructors and standard Java constructors (both their definitions and use in creating new instances). Second, `ThisClass` constructor calls must be translated to expressions that evaluate correctly even if the code is inherited to a subclass or type parameters are instantiated with different types.

Translating `ThisClass` definitions

In translating `ThisClass` constructor definitions, we wish to avoid introducing any confusion with Java constructors. That is, one should never accidentally invoke a standard constructor from a `new ThisClass` call or override a standard constructor with a `ThisClass` definition (or vice versa). A `ThisClass` constructor is translated to a standard Java constructor. In order to distinguish between the translation of a `ThisClass` and a standard constructor with identical argument types, the type signature of the `ThisClass` constructor is mangled

```
class List<T, N extends Node<T>> {
  protected @N head;

  public List() { head = null; }
  public void add(T data) {
    head = new N.ThisClass(data, head);
  }
  //pre: list is not empty
  public T remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    @N oldHead = head;
    head = head.getNext();
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 5.9: Type-safe `List<T, N>` definition

by adding an argument of type `$$MangleClass` to the end of the parameter list. This ensures that the two are not confused. For example, one can define a constructor `C()` and a constructor `ThisClass()` inside of a class `C` and guarantee that the translations of the two are distinct. The static type checker also checks that each class overrides the `ThisClass` constructors of its superclass.

Expressions `thisThisClass()` and `superThisClass()` are translated to `this()` and `super()` respectively. However, an argument of type `$$MangleClass` is added to ensure that the constructor corresponding to `ThisClass` is always called.

Translating `new ThisClass` Expressions

Translating `ThisClass` constructor definitions is relatively simple but handling `new` instance expressions is more complex. There are four legal forms of `new ThisClass` expressions:

- `new C.ThisClass(...)`
- `new C<T>.ThisClass(...)`
- `new T.ThisClass(...)  //T is a type variable`
- `new ThisClass(...)`

In particular, `ThisClass` constructors may not be invoked on objects; instead they are invoked on fully instantiated types, or the implicit fully instantiated type of `this` for `new ThisClass(...)`.

These four forms have respective return types:

- `@C`
- `@C<T>`
- `@T`
- `@ThisType` or `C<T>` or `I<T>` if called in a class `C<T>` that implements `I<T>`

`new C.ThisClass(...)`

This new instance expression is the easiest to translate. The type of the object being created is known statically, so we can simply bind the call to the specific `ThisClass` constructor called. Thus, we simply call `C`'s constructor and add an argument of type `$$MangleClass` to the end of the parameter list.

`new C<T>.ThisClass()`

This new instance expression is also simple to translate. Again, the base type of the object being created is statically known, so we can bind the call to one of `C`'s constructors. Then we simply add the `PolyClass` representing `T`, `T$$class`, and an argument of type `$$MangleClass` to the parameter list.

`new T.ThisClass(...)`

Translating `new ThisClass` expressions where the type being constructed is a type variable is more complex. As we do not know what type will be used to instantiate `T`, we can not bind the call to a specific constructor as in the previous cases. Indeed, like our translations for other run-time operations (such as `instanceof` and casts) that involve `T`, a type variable, we must use a reflective solution to invoke the correct constructor at run-time. For each `ThisClass` constructor call of the form `new T.ThisClass(...)` that is called in the body of a class, the compiler inserts a synthetic method to handle the reflective call to the constructor.

This method has name `T$$ThisClass` for a type variable `T` of a class `C`; a sample method is shown in Figure 5.10. The `T$$ThisClass` method simply looks up the constructor with the correct arguments from the base `Class` of `T$$class`. This is `T`'s `ThisClass` constructor. The correct argument types are then passed as arguments to the constructor call: the `PolyClass` objects representing `T`'s instantiations – obtained by calling `T$$class.getParams()`, and

an argument of type `$$MangleClass`. Primitive types are boxed up in their object equivalents before being passed to the `newInstance` method (the reflection API unboxes them automatically). With this method, we can translate

```
new T.ThisClass(...)
```

to a method call

```
(B)T$$ThisClass(..., T$$class)
```

The cast to `B` converts the result to `T`'s bound (also `T`'s erasure).

A few notes of explanation are needed for this translation. First, we pass the `PolyClass` `T$$class` to the `T$$ThisClass` method rather than just using the instance variable of the same name because the expression

```
new T.ThisClass(...)
```

may occur in a method with a local type parameter named `T`. In this case, `T`'s type is not stored in the instance variable, but in the method argument named `T$$class`. `T`'s type is always represented by the identifier expression `T$$class`, thus we pass it to the wrapper method. Secondly, the return type of `T$$ThisClass` is declared as `Object` because a method and a class may both declare a type variable `T` with different bounds that contain a `ThisClass` constructor with identical arguments. We can use the same `T$$ThisClass` method to perform the reflective call (their bodies would be identical if we implemented them in separate methods), but the actual returned types might be incompatible types in the translation depending on the bounds of type variables. Thus, we must define the return type as the most general type – `Object`.

`new ThisClass(...)`

The last kind of legal `new ThisClass` expression is `new ThisClass(...)`. This constructor invokes the `ThisClass` constructor with the specified argument types in the currently executing object. As a result, it can not be bound to a specific constructor because the `new` instance call might be part of a method that is inherited by a subclass. In that case, we wish to call the subclass's `ThisClass` constructor, not the superclass's. Thus, we introduce an instance method named `this$$ThisClass` for each `ThisClass` constructor definition in a class. The body of each `this$$ThisClass` method calls the corresponding translated `ThisClass` constructor, passing along the `PolyClass` objects representing the instantiations of type parameters of the enclosing object. A sample `this$$ThisClass` method is shown in Figure 5.11. By wrapping the actual constructor called in a method, we hide both the actual constructor called and the instantiations of type parameters. Thus, we can replace an expression

```
//T is a type parameter with bound B
//B contains a constructor with signature
//   ThisClass(Object, int)
private Object T$$ThisClass(Object o, int i, PolyClass T$$class) {
  //assemble array of Class types, args for constructor
  Class[] types = new Class[3 + T$$class.getParams().length];
  Object[] args = new Object[types.length];
  types[0] = Object.class;
  args[0] = o;
  types[1] = Integer.TYPE;
  args[1] = new Integer(i);
  for (int i = 0; i < T$$class.getParams().length; i++) {
    Class[i + 2] = PolyClass.class;
    args[i + 2] = T$$class.getParams()[i];
  }
  types[types.length - 1] = $$MangleClass.class;
  args[types.length - 1] = null;

  try {
    //look up constructor
    Constructor c = T$$class.getBase().getConstructor(types);
    //invoke it
    return c.newInstance(args);
  } catch (Exception e) {
    throw new ThisClassException();
  }
}
```

Figure 5.10: `T$$ThisClass` method

```
new ThisClass(...)
```

with a method call to `this$$ThisClass`

```
(C)this$$ThisClass(...)
```

Even if this expression is inherited by a subclass, the right constructor is always called because the correct `this$$ThisClass` method is invoked by dynamic dispatch. For example, if the above `new ThisClass(...)` call occurs in a class `C<T, U>`, then the returned object has type `@C<T, U>`. However, when a subclass `D<V>` executes the inherited method call, the overridden `this$$ThisClass` method invokes `D`'s constructor. Thus, the returned object has type `@D<V>`.

The declared return type of `this$$ThisClass` methods is `Object`. However, the compiler inserts a cast to the correct type when the method is actually called. If a `ThisClass` constructor includes a parameter whose type is a type variable `T` or `ThisType`, then a bridge method may have to be inserted for the `this$$ThisClass` method. The procedure for inserting the bridge method follows exactly from the techniques illustrated previously.

5.3 Type-caching Class Files

The final new platform feature added in this thesis is support for type-caching class files. This is not related to the expressiveness of the type system in `Rupiah` but rather is a simple feature to support separate compilation. Erasure is a lossy operation, thus, compiled class files do not contain the full type information for a class. Therefore, we store the necessary type information in a bytecode Signature `attribute` field so that the `Rupiah` compiler can reconstruct type information from binary class files. This feature allows separate compilation of `Rupiah` class files.

5.3.1 Motivation: Type-caching Class Files

While the basic motivation for adding bytecode `attributes` is accurately described above, it makes sense to illustrate the problem that is addressed by adding `attributes` more explicitly. The `Rupiah` compiler depends on having certain kinds of information available when compiling classes. It internally maintains information about type variables and their bounds, parameterized class types, exact types, and `ThisType` expressions. Consider the translation of the `List<T>` (we revert to this simpler list rather than using `List<T, N>` for this example), shown in Figures 5.12 and 5.13. The source file contains information about type variables and bounds, while the translated version doesn't contain either. If another class uses `List<T>`, the compiler can not determine which types were originally parameterized types or type variables. For example, the `remove()` method, which had return type `T` in the source file, has return type `Object` in the translation. The compiler

```
//Rupiah version
public class C<T, U> {
  public ThisClass(int i, Object o) {
    ...
  }
  ...
}

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

  public C(int i, Object o, PolyClass T$$class$,
    PolyClass U$$class$, $$MangleClass m) {
    ...
  }

  private Object this$$ThisClass(int i, Object o) {
    return new C(i, o, T$$class, U$$class, ($$MangleClass)null);
  }
  ...
}
```

Figure 5.11: `this$$ThisClass` constructor method

```
public class List<T> {
  protected Node<T> head;

  public List() { head = null; }
  public void add(T data) {
    head = new Node<T>(data, head);
  }
  //pre: list is not empty
  public T remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    Node<T> oldHead = head;
    head = head.getNext();
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 5.12: Rupiah source for `List<T>`

has no mechanism for determining which types were originally `Rupiah` types. Information about type parameters, bounds, exact types, and `ThisType` are lost in the translation. Thus, without bytecode Signature `attributes`, `Rupiah` source files must be compiled together.

Storing extra type information in `attribute` fields in the class file solves this problem. This technique is borrowed from GJ, and is described fully in [GBW98]. The GJ compiler stores extra information about extended types in each class file that it produces so that it can reconstruct complete class definitions from binary class files. The Java Virtual Machine Specification [LY96] allows `attribute` fields in the class file format:

> Compilers for Java source code are permitted to define and emit `class` files containing new attributes in the `attributes` tables of `class` file structures. Java Virtual Machine implementations are permitted to recognize and use new attributes found in the `attributes` tables of `class` file structures ... Java Virtual Machine implementations are required to silently ignore attributes they do not recognize.

Thus, a compiler such as the GJ compiler may insert extra `attribute` fields into the class files it generates that are ignored by other compilers and JVMs. GJ inserts type information about classes and members (instance variables, static members, and methods) in a "Signature" attribute field. Separate compilation of source files is then possible because full type information can be reconstructed from the Signature attributes in binary class files.

```
public class List {
  private PolyClass T$$class;
  protected Node head;

  public List(T$$class$) {
    T$$class = T$$class$;
    head = null;
  }
  public void add(Object data) {
    head = new Node(data, head, T$$class);
  }
  //pre: list is not empty
  public Object remove() throws EmptyListException {
    if (empty()) { throw new EmptyListException(); }
    Node oldHead = head;
    head = head.getNext();
    return oldHead.getData();
  }
  public boolean empty() { return (head == null); }
}
```

Figure 5.13: Rupiah translation of `List<T>`

5.3.2 Implementation: Type-caching Class Files

The implementation of bytecode `attributes` in `Rupiah` is straight-forward. When the compiler writes out bytecode for a source class, it stores the `attribute` fields in the class file. When a binary class file is accessed by the compiler, the information stored in `attribute` fields is immediately read and stored with the appropriate internal compiler structures.

The structure of the Signature `attribute` used in `Rupiah` is a modification of the Signatures defined for GJ classes. A specification of the Signature `attribute` is given in Figure 5.14. GJ's existing `attribute` syntax is extended to also represent exact types, `ThisType` and the exact interface of a class. Thus, from `signature` attributes, the compiler can reconstruct full type information about class members. Note that Signature `attributes` must only be stored for class members. Type information about local method variables is not needed because other classes can only interact with top-level members. Only type expressions that are accessible to other classes have Signature `attributes` stored with them.

```
MethodOrFieldSignature ::= TypeSignature

ClassSignature         ::= ParameterPartOpt super_TypeSignature
                           ExactInterfaceSignatureOpt
                           interface_TypeSignatures

ExactIntefaceSignature ::= '@Ifc' ClassTypeSignature

TypeSignatures         ::= TypeSignatures TypeSignature
                         |

TypeSignature          ::= ClassTypeSignature
                         | MethodTypeSignature
                         | TypeVariableSignature
                         | ThisTypeSignature
                         | ExactTypeSignature

ClassTypeSignature     ::= 'L' Ident TypeArgumentsOpt ';'
                         | ClassTypeSignature '.' Ident ';' TypeArgumentsOpt

ExactTypeSignature     ::= '@' TypeSignature

MethodTypeSignature    ::= TypeArgumentsOpt '(' TypeSignatures ')' TypeSignature

ThisTypeSignature      ::= 'ThisType'

TypeVariableSignature  ::= 'T' Ident ';'

TypeArguments          ::= '<' TypeSignature TypeSignatures '>'

ParameterPart          ::= '<' ParameterSignature ParameterSignatures '>'

ParameterSignatures    ::= ParameterSignatures ParameterSignature
                         |

ParameterSignature     ::= Ident ':' bound_TypeSignature
```

Figure 5.14: Rupiah Signature `attribute`

Chapter 6

Proposal Evaluations

Since Java's initial release, many have proposed extending Java with new expressive features. Yet each proposal presents different motivations, goals, and results. Indeed, because the projects start out with such varying end-goals, it is often more useful to judge the projects on the features that they hold most useful than on the actual implementations themselves. In this chapter, we offer a comparative evaluation of the strengths and weaknesses of GJ, NextGen, and `Rupiah`.

6.1 Evaluation Criteria

Previous chapters have illustrated some of the challenges of translating an extension of Java to bytecode. Many varying techniques have been suggested, yet each has a different focus. In this section we present a list of features that various projects have identified as important features for an extension of Java.

- **Homogeneous Translation:** The homogeneous translation compiles a single binary class file for each source file. This style of translation is important because it maintains Java's behavior for compiling, locating, and loading classes. Specifically, because only one class is generated to represent each parameterized class, the programmer and user of compiled programs can continue to organize and distribute classes in the same way as with Java programs.

  The homogeneous translation also has near-optimal storage efficiency because the compiled classes are essentially identical to what a Java programmer would write if

they hand-coded the same class. The only extra methods that might be needed are the bridge methods.

Finally, the pure homogeneous translation has run-time efficiency similar to a standard Java program with the same functionality. Though it relies on casts, which require a dynamic check in Java, to bridge the gap between generic representations of parameterized classes and specific instantiations, the number of casts inserted is no more than the number that would be needed in a hand-coded program. Further, these casts are proven to succeed by the extended static type system. Thus, future JVM implementations could eliminate the run-time checks for compiler-inserted casts such as these.

- **Heterogeneous Translation:** The heterogeneous translation differs from the homogeneous translation in that it compiles many classes from each source class. Rather than compiling a single generic class to represent a polymorphic class, the heterogeneous translation compiles many specific classes to represent each distinct instantiation. This results in a somewhat simpler translation (C^{++} templates use a heterogeneous translation that is as simple as macro expansion), but comes at the cost of greatly increased space and compilation efficiency. More space is needed because each instantiation of a parameterized class must be compiled to its own class. Moreover, each heterogeneous class is virtually identical (except for the internally declared types). Thus, the code bloat associated with this technique is high.

  However, despite these concerns, the heterogeneous translation is very efficient at run-time because it requires no casts. Additionally, run-time type operations execute correctly (discussed next).

- **Run-time Type Operations:** Some Java operations require full type information to execute correctly. Whether or not a project correctly translates these operations is often a quality that distinguishes it from other projects.

  - `instanceof:` The `instanceof` operator in Java allows the programmer to dynamically compare the run-time type of an object with a type. Unless parameterized classes carry some information about their actual instantiations at run-time, the `instanceof` operator can not operate correctly.
  - **Type Casts:** Correctly translating type casts involves similar issues as with `instanceof` expressions. Casts in Java are dynamically checked, and throw a `ClassCastException` if they fail at run-time. Thus, type casts for parameterized classes can not be translated correctly unless information about type variable instantiations is available at run-time.
  - **Array Types:** Similarly, `new` array expressions such as `new T[5]`, where `T` is a parameter must have access to the run-time type of `T` in order to create an array

of the correct element type. Also, arrays with parameterized element types must be able to access run-time type information about the element types to be able to execute `instanceof` and type casts correctly.

- **Full reflection for polymorphic types:** Java supports a reflection API. Thus, programmers can write classes that dynamically lookup and invoke methods, modify variables, and call constructors. Similar functionality for generic types requires run-time type information about type variables as well as a mechanism for performing reflection calls (perhaps using Java's built-in reflection).

- **Efficiency:** Evaluations of the run-time efficiency of translations are closely tied to the translation technique employed. Yet, at least two properties are desirable:
  - Programs that do not make use of parametric polymorphism should run as efficiently as standard Java programs.
  - Those that do use generic types should run as efficiently as programs with identical functionality written in pure Java.

- **Compatibility:** Issues of compatibility with Java are an important consideration for any extension of Java.
  - **Binary Compatibility:** Java programs are executed using a platform-independent virtual machine. Therefore, the translation for generic types should produce class files that can execute on the existing Java platform.
  - **Backward Compatibility:** Additionally, all Java programs should be legal programs in the new language and accepted by the compiler. Also, existing compiled classes should be usable by new code.
  - **Forward Compatibility:** Finally, retrofitting for existing code would be desirable. That is, we should be able to use an existing class without parameters by specifying a polymorphic interface for that class.

- **Primitive Types as Type Parameters:** It may be desirable to use the primitive types: `boolean, byte, char, double, float, int, long, short` as instantiations of type parameters.

- **Per-Instantiation Static Variables:** Static fields in some implementations can be declared with parameterized or generic types. The semantics of these fields is that one object is shared between all identical instantiations of the class (rather than between all instances of the base class).

- **Constructors for Generic Types:** Some mechanism should be provided for constructing objects whose type is a parameter.

- **Additional Type System Constructs:** While generic types greatly increase the expressiveness of the type system, more features can be helpful in order to represent the concepts that many programs require.
  - `ThisType:` Binary methods occur naturally in many programs. However, programming them safely in Java is very difficult because inheritance causes problems when a binary method is executed by a subclass. `ThisType` offers a solution by incorporating the concept of a type that is the same as that of `this` in the type system.
  - **Exact Types** Exact types are necessary to support a `ThisType` construct safely. They are also useful for explicitly programming homogeneous structures.

6.2 Language Comparisons

In this section, we evaluate GJ, NextGen, and `Rupiah` under the criteria defined above. Figure 6.1 illustrates the key features of each language in a side-by-side comparison.

6.2.1 GJ

The GJ language adds generics to Java via a pure homogeneous translation. GJ places an emphasis on compatibility with the Java language. Thus, every legal Java program is a legal GJ program, compiled GJ programs run on standard JVMs, and a mechanism is provided for outfitting existing Java classes with generic types. The homogeneous translation used in GJ is very efficient in both run-time and space efficiency. GJ does not allow primitive types as instantiations of type variables. However, Pizza, GJ's predecessor, did provide this functionality so it is reasonable to think that it could easily be extended to GJ if desired.

The biggest issue with GJ's translation is that almost all operations that require full run-time information are illegal in GJ because it uses erasure – an inherently lossy operation. The only casts and `instanceof` tests involving parameterized types that are legal in GJ are those that can be completely determined statically. Thus, they lose the ability to do any reflective operations on parameterized types. While GJ provides "raw types" as a substitute – that is one can use the erasure of a generic type in an introspective operation – no ability is provided to obtain any information about type variables and their instantiations at run-time. Additionally, no mechanism is provided for constructing objects whose type is a type variable.

While GJ provides a solid implementation of an extension of Java with generic types, it suffers from a complete lack of support for operations that require run-time type information. However, the close compatibility (binary, source, and retrofitting) that GJ enjoys with Java is an important and useful feature.

	GJ	NextGen	Rupiah
Translation Technique			
Homogeneous	√		√
Heterogeneous		√	
Supported Run-time Operations			
`instanceof`		√	√
Type Casts		√	√
`new` Array Expressions		√	√[a]
Reflection for Generic Types			
Efficiency			
Run-Time	√	√	
Space	√		√
Compilation	√		√
Compatibility			
Binary	√	√	√
Backward	√	√	√[b]
Forward	√	√	
Miscellaneous Features			
Primitive types as Instantiations			
Per-Instantiation Static Variables		√	
Constructors for Generic Types		√[c]	√
`ThisType`			√
Exact Types			√

[a] Not supported in the current compiler, see Appendix C for more details.
[b] The proposal in Appendix C would break source compatibility for some Java programs.
[c] It is unclear how `new T()`, where `T` is a type variable, is statically checked in NextGen.

Figure 6.1: Comparing Language Features

6.2.2 NextGen

The NextGen language aims to correct the limitations of GJ in operations that require run-time type information. The solution used in NextGen is a hybrid translation that uses aspects of the homogeneous and heterogeneous translations. NextGen harnesses the nice space properties of the homogeneous translation, by compiling a base class containing the bulk of the program code and wrapper classes to represent run-time instantiations of parameterized classes. The result is a language where all types are compatible with Java semantics for operations that require run-time type information. Type casts, `instanceof`, and `new` array expressions all operate correctly in NextGen because the full type information is implicitly carried in the heterogeneous class. NextGen does not define a reflection API for parameterized types. However, one could be devised from the specified translation as the full run-time information about instantiations is present.

However, though the hybrid translation cuts down on code bloat, it still compiles many different classes for each source – differing from Java's normal compilation behavior. In particular, using an instantiation for the first time results in a new compiled class. Additionally, though the space efficiency of NextGen is much improved over a pure heterogeneous translation, the number of classes generated for a commonly used class can result in a significant amount of required space. Finally, though NextGen has minimal support for constructors of the form `new T()`, constructor calls for generic types that include arguments are not supported.

6.2.3 Rupiah

`Rupiah` supports parametric polymorphism by a homogeneous translation that is similar to GJ. However, it also uses Java's reflection API to associate information about type variable instantiations with polymorphic classes. With this information, `Rupiah` is able to generate code that properly executes Java's introspective operations: `instanceof`, type casts, and `new` array expressions.[1] Additionally, `Rupiah` adds several new language features that significantly increase the expressiveness of the language. These features include `ThisType`, exact types, and `ThisClass` constructors. The other languages evaluated here do not explore other expressive features beyond generic types. We feel that added expressiveness is important – while generic types go a long way to making static type safety possible for some programs, they are not enough in many situations. Because `Rupiah` supports these features, many more programs can be written within the bounds of its static type system.

While `Rupiah` uses a homogeneous translation that is very efficient in space and compilation efficiency, support for run-time type operations uses Java's reflection to access the instantiations of generic types. Thus, the run-time efficiency of `Rupiah` programs is somewhat less than a pure Java program. However, because `Rupiah` provides additional

[1] Not yet implemented, see Appendix C.

functionality, including stronger guarantees of type safety and the ability to perform introspective operations on generic types, some decreased efficiency is expected. However, the programmer does not pay a large price in run-time performance unless they use the new features. If the programmer avoids costly statements like `object instanceof T`, where `T` is a type parameter, then the expected performance is similar to GJ. The only overhead is creating and assigning `PolyClass` objects.

`Rupiah` is binary-compatible with the Java platform. That is, compiled classes run on a standard JVM. Java source programs in the current implementation are also legal `Rupiah` programs. However, the implementation of the array proposal in Appendix C would break compatibility with Java programs that use the unsafe covariant array subtyping rule. This unfortunate fact illustrates the long-term effects that a hole in the static type system can cause. Though Java patches this hole with a run-time check, it causes serious problems for the `Rupiah` project. Retrofitting is not possible with `Rupiah` programs essentially because constructors can not be retrofitted with extra `PolyClass` arguments. We could conceivably allow retrofitting in the manner that GJ does, however we would lose our guarantees that run-time type information is always available for operations that require it. This problem stems from the fact that we can not insert `PolyClass` instance variables into a pre-compiled classes such as `Vector`, nor can we add helper methods like `instanceof$Vector`. Thus, we tradeoff ease of migration for more consistent run-time semantics. `Rupiah` does not currently support instantiations of type variables with primitive types nor does it support per-instantiation static variables. `Rupiah`'s `ThisClass` constructors provide a robust mechanism in for creating objects whose type is a type variable or `ThisType`.

6.3 Overall Evaluation

No single project implements all of the desirable features described in Section 6.1. Thus, to evaluate them we must examine some of the tradeoffs implicit in each approach. We prefer `Rupiah` to the other two languages both for its closer adherence to Java philosophies and for the additional expressiveness gained from the features beyond parametric polymorphism.

Evaluating how closely a language fits with Java is a subjective task. GJ proponents argue that their implementation, which provides the closest compatibility of any other language, makes it the most consistent with Java philosophies. The natural rejoinder to that assertion is that GJ's inconsistencies with Java semantics for operations that require run-time type information result in a significant gap between the languages.

Similarly, NextGen supporters argue that their language is a close fit with Java because they both support retrofitting and run-time compatible types. However, the fact that their language is compiled heterogeneously causes compatibility issues because programmers lose the property that one source file is represented by one compiled class file.

`Rupiah` achieves compatibility with Java semantics for run-time operations such as

`instanceof` and checked casts while remaining a homogeneous translation. We argue that it is in fact more closely tied to Java philosophies than either GJ or NextGen. While the retrofitting features that GJ offers are important – certainly providing a migration mechanism is an important feature to help a language gain adoption – it does not affect the core of the language itself. Thus, the properties that we find lacking in GJ are more significant in our eyes than the smooth forward compatibility track that they provide.

Finally, `Rupiah`'s other type system features should not be ignored. `ThisType` incorporates a concept that allows programs to be expressed that simply could not be written safely with F-bounded polymorphism alone. Binary methods occur often enough that supporting them in the type system offers great benefits to the programmer. Our sample programs in Appendix B serve as examples as to how beneficial it can be to write programs with the features included in `Rupiah`. Finally, `ThisClass` constructors address a real problem with parametric polymorphism – namely that constructing generic types is difficult to do without escaping the static type system. Thus, we hold that `Rupiah` affords the most practical safety to programmers.

6.3.1 Sun Microsystems Proposal

Recently, Sun Microsystems has formally explored the possibility of adding generics to Java. A paper published by the expert group [BCK+01] indicates that generics will most likely be incorporated into Java in the near future. The direction that Sun is taking seems to be closely related to the GJ project. Indeed, the current 1.3 `javac` is based on Odersky's GJ compiler (with support for generic types removed). Though we are concerned about GJ's lack of support for operations that require run-time type information, we are very pleased to see generics entering the language in the near future. Allowing programmers to get comfortable programming with generic types is an important step in the right direction. Importantly, the proposal is open-ended enough that future developments, for example a modified JVM, could be implemented to fix some of the current limitations with GJ.

Chapter 7

Future Work

We present some suggestions for future directions for the `Rupiah` project. With this thesis, a direct translation to bytecode for the `Rupiah` project seems thoroughly explored. It is clear that such a translation is feasible, yet some concerns with the run-time efficiency of the language and with array types have led us to suggest a different style of translation based on a modified JVM. A translation that includes JVM modifications, as well as some proposed additional language features provide significant opportunities for future work.

7.1 The Array Problem

A portion of `Rupiah` that leaves us feeling uneasy is the support for arrays in the language. The fact that arrays are objects created by the JVM rather than by constructors means that the standard `Rupiah` technique of associating `PolyClass` objects with each polymorphic object can not be performed for arrays. The solution described in Appendix C is not yet implemented, and is unsettling because it breaks source compatibility with Java programs that make use of Java's unsafe covariant array subtyping rule. Further, because it involves storing arrays in wrapper classes, it raises major concerns about run-time efficiency.

7.2 Efficiency Issues

A second related concern is that the `Rupiah` translation, which is based on erasure, is not very run-time efficient. Some parts of the translation, including most operations involving type parameters, unavoidably rely on reflection to operate correctly. However, there

are some simple solutions that can be used to greatly increase the efficiency of `Rupiah` programs. Implementing the `PolyClassLoader` proposal [VN00], described in Section 4.5, would greatly reduce the number of `PolyClass` objects created for `Rupiah` programs as well as making equality checks for `PolyClass` objects much more efficient. It is a relatively simple solution, thus we think that it should be implemented in the `Rupiah` compiler soon.

7.3 All Roads Lead to the JVM

Our homogeneous translation based on reflection might be inherently better served with a few small changes to the JVM. Some projects, including [SA98], have implemented small JVM changes that greatly simplify translating polymorphic programs. The simplest JVM change that would help to support parametric polymorphism would be to modify the `Class` file so that it contained information about type parameters. This would allow all generic types including array types, to carry their instantiation information at run-time. Also, it would make implementing a reflection API for polymorphic types as easy as extending the current implementation in a few key places to be aware of type parameters. Even retrofitting could be easily supported because we no longer need to pass `PolyClass` objects to constructors for run-time type compatibility. Recall that `Rupiah` can not support retrofitting because we do not have a mechanism for inserting `PolyClass` objects into the argument lists of constructors for pre-compiled classes. With a modified JVM, the interpreter could manage the implicit type parameters for retrofitted classes instead. Finally, the representation of type parameters would be managed by the JVM so we could expect significant speedups over the `Rupiah` translation. Rather than needing complicated translations for `instanceof`, casts, et cetera, we could simply use the built in Java bytecode instructions.

The gains of being able to properly support arrays with generic element types may be enough to merit moving towards JVM modifications. While the proposal in Appendix C is plausible, the sacrifices made to support arrays in this fashion are great. JVM modifications would simplify our translation greatly, especially for arrays, as well as restoring full source compatibility with Java. Therefore, we think that it has significant merit.

7.4 Miscellaneous Features

Several features that we would like to support are not currently implemented in the `Rupiah` language.

- **Prove Type Safety of `ThisClass`:** While most of the features in `Rupiah` have been proven type safe, either in GJ or $\mathcal{LOOM}$, we would like to also give a proof that `ThisClass` is sound. Experience has shown us that even proving simple concepts can

lead to much broader revelations; and that even proofs that appear trivial on the surface can in fact require much effort. The novel concept of `ThisClass` constructors needs formalization in order to use it with real confidence.

- **Better support for static members:** With the addition of method-level type parameters, we no longer need to restrict the use of class type variables to non-static members.

  Instead we can translate static methods as if they had a local type variable for each type parameter to the enclosing class. Thus, the a method `void foo()` in a class `C<T>` could be translated to

  ```
  static void <T> foo();
  ```

  Whenever `foo()` is invoked, in an expression like

  ```
  C<Integer>.foo();
  ```

  the full instantiated type of `C` is specified, thus `T` is known and can be translated appropriately. For example, the above method call would become:

  ```
  C.<Integer>foo();
  ```

  This, to support the use of type variables in static methods, we could translate class type variables to local method type variables.

  Similarly, we should implement per-instantiation static variables as suggested by Burstein [Bur98]. Following his suggestion, we would use a hashtable to store a common object for each instantiation of a class. For each static field in a class, we would add `get` and `set` methods to access the field, passing the instantiation of type variables as a key to the hashtable. This would allow static variables that involve type parameters that are shared between like instantiations.

- **Support for more Expressive Bounds:** The type parameter declaration `<T extends ClassType, U extends T>` is not currently supported in `Rupiah` or GJ because bounds must be reference types. The rationale is that bounding `U` by `T` is no different from constraining it with `ClassType`. However, we can imagine an application where being able to specify a type parameter that must be at least as specific as another type parameter could be useful. In the current implementation of `Rupiah` for example, if `T` and `U` were both bounded by `ClassType`, then `T` might be instantiated with a more specific type than `U`. No mechanism currently exists that `U` must be no more general than `T`.

A second feature we would like to see implemented in `Rupiah` is the ability to specify an interface type as a bound that can only be instantiated with a class type. This feature should be relatively easy to support because it only involves a simple type-checking rule. We propose that the `extends` keyword in bound declarations continue to indicate that any type that matches the bound may be used to instantiate it. The `implements` keyword meanwhile may only be used with bounds that are interface types and specifies that only a matching class type may be used to instantiate that variable. This functionality would also allow us to support `ThisClass` declarations in interfaces. Currently, `ThisClass` declarations may not appear in interfaces because that interface can not be constructed. Thus, if one wishes to construct an object whose type is a type variable, the bound of that type variable must be a class type that contains a `ThisClass` constructor. However, if interfaces could be used as bounds that can only be instantiated by class types, then it might be reasonable to allow `ThisClass` declarations in interfaces so that they can be used in bounds.

7.5 Wrapup

The `Rupiah` project incorporates several important features to the Java programming language. Match-bounded parametric polymorphism, exact types, `ThisType`, and `ThisClass` constructors all increase the expressiveness of the static type system and allow many more programs to be written safely than in standard Java. This increased measure of safety that can be ascertained for `Rupiah` programs provides significant benefit to the programmer because her programs are less likely to suffer from run-time errors.

Implementing these features has been an interesting and challenging research project. Following the lead of other related projects, we have explored a translation to bytecode. The homogeneous translation used in `Rupiah` has many nice properties including consistency with Java semantics for compilation, optimal space requirements, and reasonable run-time efficiency. However, translating in such a fashion that operations that require run-time type information operate consistently for generic types has proven more difficult.

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*, page to appear, 1997. 38

[Bai98] Duane A. Bailey. *Java Structures: Data structures in Java for the principled programmer*. McGraw-Hill, 1998. 109

[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. 14, 17

[BCK+01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stautamire, Kresten Thorup, and Philip Wadler. Adding Generics to the Java Programming Language: Participant Draft Specification. Draft Paper, April 2001. 98

[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. i, 5, 17, 18

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM. i, 5, 24

[Bru93] K. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 285–298, 1993. 5, 17

[Bru97] Kim B. Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Technical report, Williams College, 1997. 5

[Bru98] Kim B. Bruce. Building an Extensible Interpreter for a Simple Functional Language. Message to Java-generics electronic mail list, August, 1998. 114

[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. 5, 17

[Bur98] Jon Burstein. *Rupiah: An extension to Java supporting match-bounded parametric polymorphism, ThisType, and exact typing.* Williams College Senior Honors Thesis, 1998. 5, 41, 101

[BvG93] Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. Technical report, Williams College, 1993. 5

[Car96] L. Cardelli. Bad engineering properties of object-oriented languages. *ACM Computing Surveys*, 28(4es):150, December 1996. 4

[Car97] Robert Cartwright. Challenge to build an extensible interpreter. Message to Java-generics electronic mail list, 1997. 114

[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. i, 14, 24

[CS98] R. Cartwright and G. Steele, Jr. Compatible genericity with run-time types for the Java programming language, 1998. 5, 24, 36

[GBW98] David Stoutamire Gilad Bracha, Martin Odersky and Philip Wadler. GJ specification. draft paper, May 1998. 87, 106

[GJS96] James Gosling, Bill Joy, and Guy Steele, Jr. *The Java Language Specification.* Addison Wesley, 1996. 1, 3, 11, 35, 106

[GRJV94] E. Gamma, R.Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and reuse of object-oriented designs.* Addison-Wesley, 1994. 115

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996. 87

[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. 38

[Mey92] B. Meyer. *Eiffel: the language.* Prentice-Hall, 1992. 17

[ORW97] Martin Odersky, Enno Runne, and Philip Wadler. Two ways to bake your pizza – translating parameterised types into Java. Technical Report CIS-97-016, University of South Australia, 1997. 24

[SA98] Jose H. Soloranzo and Suad Alagić. Parametric polymorphism for Java: A reflective solution. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM. 38, 100

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *OOPSLA '86 Proceedings*, pages 9–16. ACM SIGPLAN Notices,21(11), November 1986. 17

[VN00] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Minneapolis, October 2000. ACM. 38, 65, 100

[Wal96] James Waldo. Interface and implementation [Java]. *UNIX review*, 14(13):81–82, 85–86, December 1996. 58

Appendix A

Rupiah Grammar

A grammar defines the syntax and parsing for a programming language. We give a grammar for **Rupiah**, based on a modification of GJ's grammar, as defined in [GBW98]. Section numbers correspond to those in [GJS96].

A.1 Generic Types

§4.3 Reference Types and Values

```
ReferenceType               ::= NonExactType
                             |  ExactType

NonExactType                ::= ClassOrInterfaceType
                             |  ArrayType
                             |  TypeVariable
                             |  ThisType

ExactType                   ::= @ NonExactType

TypeVariable                ::= Identifier

ClassOrInterfaceType        ::= Name
                             |  Name < ReferenceTypeList1

ReferenceTypeList1          ::= ReferenceType1
                             |  ReferenceTypeList , ReferenceType1
```

```
ReferenceType1              ::= ReferenceType >
                             |  Name < ReferenceTypeList2

ReferenceTypeList2          ::= ReferenceType2
                             |  ReferenceTypeList , ReferenceType2

ReferenceType2              ::= ReferenceType >>
                             |  Name < ReferenceTypeList3

ReferenceTypeList3          ::= ReferenceType3
                             |  ReferenceTypeList , ReferenceType3

ReferenceType3              ::= ReferenceType >>>

TypeParametersList          ::= TypeParamterList , TypeParameter
                             |  TypeParameter


TypeParameters              ::= < TypeParameterList1

TypeParameterList1          ::= TypeParameter1
                             |  TypeParameterList1 , TypeParameter1

TypeParameter1              ::= TypeParameter >
                             |  TypeVariable extends ReferenceType2

TypeParameter               ::= TypeVariable TypeBoundOpt

TypeBound                   ::= extends ClassOrInterfaceType
```

§8.1, 9.1 Class and Interface Declarations

```
ClassDeclaration            ::= ModifiersOpt class Identifer
                                TypeParametersOpt SuperOpt InterfacesOpt ClassBody

InterfaceDeclaration        ::= ModifiersOpt interface Identifier TypeParametersOpt
                                ExtendsInterfacesOpt InterfaceBody
```

A.2 Polymorphic Methods

§8.4 Method Declarations

```
MethodHeader                ::= ModifiersOpt TypeParametersOpt Type
```

```
                                MethodDeclarator ThrowsOpt
                              | ModifiersOpt TypeParametersOpt void
                                MethodDeclarator ThrowsOpt
```

§15.12 Method Invocation Expressions

```
MethodInvocation              ::= MethodExpr (ArgumentListOpt )

MethodExpr                    ::= MethodId
                                | Primary . MethodId
                                | super . MethodId

MethodId                      ::= TypeArgumentsOpt MethodName
```

A.3 Constructors

§8.8 Constructor Declarations

```
ConstructorDeclaration        ::= ModifiersOpt ConstructorDeclarator
                                  ThrowsOpt ConstructorBody

ConstructorDeclarator         ::= SimpleTypeName ( FormalParameterListOpt  )
                                | ThisClass ( FormalParameterListOpt  )
```

§8.8.5.1 Explicit Constructor Invocations

```
ExplicitConstructorInvocation ::= this ( ArgumentListOpt ) ;
                                | super ( ArgumentListOpt ) ;
                                | superThisClass ( ArgumentListOpt ) ;
                                | thisThisClass ( ArgumentListOpt ) ;
                                | Primary . super ( ArgumentListOpt ) ;
```

§15.9 Class Creation Expressions

```
ClassInstanceCreationsExpression ::= new Name TypeArgumentsOpt ( ArgumentListOpt )
                                   | new ThisClass ( ArgumentListOpt )
                                   | new TypeVariable . ThisClass ( ArgumentListOpt )
                                   | new ClassType . ThisClass ( ArgumentListOpt )
```

Appendix B

Sample Rupiah Programs

Language features are often best motivated by examples. Here we give the source code for several useful Rupiah programs. It is our hope that by seeing some of the expressive features as used in real programs, the benefits of these new features will become clear.

B.1 Lists in Rupiah

These lists are motivated by the structures package [Bai98]. We give the full definitions for doubly and singly linked lists, as well as all of the relevant interfaces. The features demonstrated in these programs are bounded parametric polymorphism, exact types, ThisType, and ThisClass constructors. Besides the added type safety gained by using these constructs, we are also able to reuse much more code.

```
/*************************************************************/
public interface Store {
    public int size();
    public boolean isEmpty();
    public void clear();
}
/*************************************************************/
public interface Collection<T> extends Store {
    public boolean contains(T value);
    public void add(T value);
    public T remove(T value);
    public Iterator<T> elements();
```

```
}
/*************************************************************/
public interface List<T> extends Collection<T> {
    public void addToHead(T value);
    public void addToTail(T value);
    public T peek();
    public T tailPeek();
    public T removeFromHead();
    public T removeFromTail();
}
/*************************************************************/
public interface Enumeration<T> {
    public boolean hasMoreElements();
    public T nextElement();
}
/*************************************************************/
public interface Iterator<T> extends Enumeration<T> {
    public void reset();
    public T value();
}
/*************************************************************/
public class Node<T> {
    protected T data;
    protected @ThisType nextElement;

    public ThisClass(T v, @ThisType next) {
        data = v;
        nextElement = next;
    }
    public ThisClass(T v) { thisThisClass(v, null); }

    public @ThisType right() { return nextElement; }
    protected void setNext(@ThisType next) { nextElement = next; }
    public void setRight(@ThisType right) { setNext(right); }
    public T value() { return data; }
    public void setValue(T value) { data = value; }
}
/*************************************************************/
public class DblNode<T> extends Node<T> {
    protected @ThisType previousElement;

    public ThisClass(T v, @ThisType next, @ThisType previous) {
        superThisClass(v, next);
        setLeft(previous);
    }
```

```
    public ThisClass(T v, @ThisType next) { thisThisClass(v, next, null); }
    public ThisClass(T v) { thisThisClass(v, null); }

    public @ThisType left() { return previousElement; }
    protected void setRight(@ThisType right) {
        setNext(right);
        if (right != null) right.setPrevious(this);
    }
    protected void setPrevious(@ThisType prev) {
        previousElement = prev;
    }
    public void setLeft(@ThisType previous) {
        if (previous != null) previous.setRight(this);
        else setPrevious(null);
    }
}
/*************************************************************/
public class EmptyListException extends Exception { }
/*************************************************************/
public class LinkedList<T, N extends Node<T>> implements List<T> {
    protected int count;
    protected @N head;
    protected @N tail;

    public LinkedList() {
        head = tail = null;
        count = 0;
    }

    public void add(T value) { addToHead(value); }
    public void addToHead(T value) {
        head = new N.ThisClass(value, head);
        if (tail == null) tail = head;
        count++;
    }
    public T removeFromHead() throws EmptyListException {
        if (empty()) throw new EmptyListException();
        @N temp = head;
        head = head.right();
        if (head == null) tail = null;
        temp.setRight(null);
        count--;
        return temp.value();
    }
    public void addToTail(T value) {
```

```
        @N oldTail = tail;
        tail = new N.ThisClass(value);
        if (head == null) head = tail;
        else {
          oldTail.setNext(tail);
        }
        count++;
    }
    public T removeFromTail() throws EmptyListException {
        if (empty()) throw new EmptyListException();
        @N oldTail = tail;
        if (size() == 1) head = tail = null;
        else {
            @N newTail = head;
            for(; newTail.right() != oldTail; newTail = newTail.right());
            tail = newTail;
            tail.setNext(null);
        }
        count--;
        return oldTail.value();
    }
    public T peek() throws EmptyListException {
        if (empty()) throw new EmptyListException();
        return head.value();
    }
    public T tailPeek() throws EmptyListException {
        if (empty()) throw new EmptyListException();
        return tail.value();
    }
    public boolean contains(T value) {
        @N finger = head;
        while (finger != null && !finger.value().equals(value)) {
            finger = finger.right();
        }
        return finger != null;
    }
    public T remove(T value) {
        @N finger = head;
        @N previous = null;
        while (finger != null && !finger.value().equals(value)) {
            previous = finger;
            finger = finger.right();
        }
        if (finger != null) {
            if (previous != null) previous.setRight(finger.right());
```

```
            else head = finger.right();
            if (tail == finger) tail = previous;
            finger.setRight(null);
            count--;
            return finger.value();
        }
        return null;   //no element equal to value
    }
    public int size() { return count; }
    public boolean isEmpty() { return size() == 0; }
    public void clear() {
        head = tail = null;
        count = 0;
    }
    public Iterator<T> elements() {
        return new LinkedListIterator<T, N>(head);
    }
}
/*************************************************************/
public class LinkedListIterator<T, N extends Node<T>> implements Iterator<T> {
    protected @N current;
    protected @N head;

    public LinkedListIterator(@N t) {
        head = t;
        reset();
    }

    public void reset() { current = head; }
    public boolean hasMoreElements() { return current != null; }
    public T nextElement() {
        T temp = current.value();
        current = current.right();
        return temp;
    }
    public T value() { return current.value(); }
}
/*************************************************************/
public class DblLinkedList<T, N extends DblNode<T>>
    extends LinkedList<T, N> {
    public DblLinkedList() {
        head = tail = null;
        count = 0;
    }
```

```
    public T removeFromHead() {
        T retVal = super.removeFromHead();
        head.setLeft(null);
        return retVal;
    }
    public void addToTail(T value) {
        tail = new N.ThisClass(value, null, tail);
        if (head == null) head = tail;
        count++;
    }
    public T removeFromTail() {
        if (empty()) throw new EmptyListException();
        @N temp = tail;
        tail = tail.left();
        if (tail == null) head = null;
        else tail.setRight(null);
        count--;
        return temp.value();
    }
    public Interator<T> reverseElements() {
      return new LinkedListReverseIterator<T, N>(head);
    }
}
/*************************************************************/
public class LinkedListReverseIterator<T, N extends DblNode<T>>
   extends LinkedListIterator<T, N> {

    public LinkedListIterator(@N t) {
        head = t;
        reset();
    }
    public T nextElement() {
        T temp = current.value();
        current = current.left();
        return temp;
    }
}
```

B.2 An Extensible Interpreter

In [Bru98], Bruce describes a technique for programming an extensible interpreter. This technique is based on a suggestion by Cartwright [Car97]. Under this scheme, one can write an interpreter for a language, and have the ability to add new language constructs

without re-writing any of the existing code. A new interpreter can be implemented by simply providing functionality for the new concepts. The design is based on the Visitor pattern [GRJV94].

Here, we show the interpreter suggested by Bruce as implemented in `Rupiah`. The (basically functional) source language to be interpreted is defined by `LangProcessor<Result>`, which supports four kinds of expressions:

- `constCase` – constants
- `varCase` – variables
- `lamCase` – lambda expressions
- `appCase` – function application

The extended language is defined by `ExtLangProcessor<Result>` and includes additional support for `plusCase` – simple arithmetic addition expressions.

Note that polymorphic methods, `ThisType` and `ThisClass` constructors are needed to implement this solution.

This code is not meant to demonstrate a robust, industry-strength interpreter but rather serve as a proof-of-concept that an extensible interpreter could be easily programmed using the features introduced in `Rupiah`.

```
//any values
interface Val { }

//an environment -- binds identifiers to values
interface Env {
  Env nullEnv();
  Env extend(String var, Val value);
  Val lookup(String var);
}

//original language constructs
interface LangProcessor<Result> {
  Result constCase(ConstForm cf);
  Result varCase(VarForm vf);
  Result lamCase(LamForm lf);
  Result appCase(AppForm af);
}

//extended language construct
interface ExtLangProcessor<Result>
          extends LangProcessor<Result> {
```

```
  Result plusCase(PlusForm pf);
}

//abstract syntax for languages
interface Form <Processor extends LangProcessor> {
  <Result, Visitor extends Processor<Result>> Result process(Visitor lp);
}
/*********************************************************************/
class IntVal implements Val {
  int value;
  IntVal(int value) { this.value = value; }
}
class Closure implements Val {
  Env env;
  LamForm fun;
  Closure(LamForm fun, Env env) {
        this.fun = fun; this.env = env; }
}
/*************************************************************/
class ConstForm implements Form<LangProcessor> {
  int value;
  ConstForm(int value) { this.value = value; }
  <Result, Visitor extends LangProcessor<Result>> Result process(Visitor lp) {
          return lp.constCase(this); }
}
class VarForm implements Form<LangProcessor> {
  String name;
  VarForm(String name) { this.name = name; }
  <Result, Visitor extends LangProcessor<Result>> Result process(Visitor lp) {
          return lp.varCase(this); }
}
class LamForm implements Form<LangProcessor> {
  String var;
  Form body;
  LamForm(String var, Form body) {
        this.var = var; this.body = body;}
  <Result, Visitor extends LangProcessor<Result>> Result process(Visitor lp) {
          return lp.lamCase(this); }
}
class AppForm implements Form<LangProcessor> {
  Form fun, arg;
  AppForm(Form fun, Form arg) { this.fun = fun; this.arg = arg; };
  <Result, Visitor extends LangProcessor<Result>> Result process(Visitor lp) {
          return lp.appCase(this); }
}
```

```
/*************************************************************/
// interpreter for original language
class Interp implements LangProcessor<Val> {
  Env env;

  ThisClass (Env env) { this.env = env; }

  Val constCase(ConstForm, cf) { return new IntVal(cf.value); }
  Val varCase(VarForm vf) { return env.lookup(vf.name); }
  Val lamCase(LamForm lf) { return new Closure(lf,env); }
  Val appCase(AppForm af) {
    return apply(af.fun.<Val,ThisType>process(this),
                 af.arg.<Val,ThisType>process(this));
  }
  Val apply(Val rator, Val rand) {
    if (rator instanceof Closure) {
      Closure clos = (Closure) rator;
      LamForm lf = clos.fun;
      Env newenv = clos.env.extend(lf.var,rand);
      return lf.body.<Val,ThisType>process(new ThisClass(newenv));
    }
    else throw
      throw ApplicationException("non-function " + rator +
                                 " used as function");
  }
}

// additional construct in extended language
class PlusForm implements Form<ExtLangProcessor> {
  Form lhs, rhs;
  PlusForm(Form lhs, Form rhs) {
        this.lhs = lhs; this.rhs = rhs; }
  <Result, Visitor extends ExtLangProcessor<Result>> Result process(Visitor lp) {
          return lp.plusCase(this); }
}
/*************************************************************/
// interpreter for extended language
class ExtInterp extends Interp implements ExtLangProcessor<Val> {

  ThisClass (Env env) { super(env); }

  Val plusCase(PlusForm pf) {
    Val lval = pf.lhs.<Val,ThisType>process(this);
    Val rval = pf.rhs.<Val,ThisType>process(this);
    if ((lval instanceof IntVal) && (rval instanceof IntVal))
```

```
      return new IntVal(((IntVal) lval).value + ((IntVal) rval).value);
    else
      throw new PlusException("plus operator applied to " + lval + " and " +
                              rval);
  }
}
/*****************************************************************/
class ApplicationException extends RunTimeException {
  ApplicationException(String s) { super(s); }
}

class PlusException extends RuntimeException {
  PlusException(String s) { super(s); }
}
/*****************************************************************/
```

Appendix C

Array Proposal

Declarations of arrays whose element type contains a type variable, parameterized type, or `ThisType` cause problems in `Rupiah`. Recall from Chapter 4 that `Rupiah` stores a `PolyClass` object with each parameterized class. These `PolyClass` objects are used in operations that require full type information at run-time. The `PolyClass` objects are created and passed along to constructors for polymorphic classes. Arrays are also objects in Java. However, they are not constructed by calling a constructor, but by bytecode instructions. Because all array operations are implemented by bytecode instructions, we have no way to store a `PolyClass` object with an array. Thus, we have no way to correctly perform `instanceof`, casts, or `new` array expressions for arrays with the problematic element types.

C.1 Proposed Solution

Unfortunately, no clean solution is apparent for this issue. In this section, we propose a solution to the problem that allows us to properly translate array types by storing certain arrays in wrapper classes. However, the change results in some drastic changes to the type checking rules for array types, and means that some Java code – namely code that makes use of Java's unsafe covariant array subtyping rule – is not compatible with `Rupiah`. We have not yet implemented this solution in our compiler. However, we provide a casual description of the translation here to illustrate that the technique could work.

The proposed solution is to wrap each array in a separate class which stores the actual array and a `PolyClass` object representing the element type. All arrays with element types that are "`Rupiah` types" (parameterized types, type parameters, or `ThisType`) are replaced

```
package EDU.williams.rupiah;

public final class RupiahArray {
  public final Object[] data;
  public final int length;
  private PolyClass elemType;

  public RupiahArray(Object[] data, PolyClass elemType) {
    this.data = data;
    this.elemType = elemType;
    this.length = data.length; //this works: constructors can
                               //initialize final fields
  }

  //does a shallow clone of the array.
  //we just reuse the PolyClass
  public Object clone() {
    return new RupiahArray(data.clone(), elemType);
  }

  public boolean $instanceOf$(Object other) {
    return (other instanceof RupiahArray
           && ((RupiahArray)other).elemType.equals(elemType));
  }
}
```

Figure C.1: Definition of `RupiahArray`

with instantiations of this wrapper class. Operations such as `instanceof` and type casts use the `PolyClass` object stored with the object. The wrapper class is named `RupiahArray` and is defined in Figure C.1.

Each new array expression involving a `Rupiah` type is wrapped in a `RupiahArray`. For example, the declaration:

```
List<T>[] ltarray = new List<T>[5];
```

becomes

```
RupiahArray ltarray = new RupiahArray(new List[5],
                          new PolyClass(List.class,
                              new PolyClass[] { T$$class}));
```

or

```
T[] tarray = new T[5]
```

becomes

```
RupiahArray tarray = new RupiahArray(
     java.reflect.Array.newInstance(T$$class.getBase(),
                                    new int[] { 5 }),
                                    T$$class)
```

In order to be able to create arrays with element type `ThisType`, we need a run-time representation of `ThisType` that changes meaning in subclasses parallel to `ThisType` itself. To accomplish this, we store the type of `ThisType` in a `PolyClass` instance variable. For each class with an expression of type `ThisType`, a field with declaration

```
protected PolyClass $$ThisType;
```

is added to the class. The `$$ThisType` field is initialized in each constructor with a `PolyClass` object representing the exact interface that the class implements as shown in Figure C.2. This field is protected and not private, and thus is inherited by any subclasses. With this information, we can always create arrays of type `ThisType[]` by accessing the type stored in `$$ThisType`. Therefore, the expression

```
List<ThisType>[] lttarray = new List<ThisType>[5];
```

becomes

```
RupiahArray lttarray = new RupiahArray(new List[5],
                           new PolyClass(List.class,
                               new PolyClass[] { $$ThisType}));
```

or

```
ThisType[] ttarray = new ThisType[5]
```

becomes

```
RupiahArray tarray = new RupiahArray(
     java.reflect.Array.newInstance($$ThisType.getBase(),
                                    new int[] { 5 }),
                                    $$ThisType)
```

Accessing an element of a `RupiahArray` translates as an access to the `data` field of `RupiahArray`. For example the statement:

```
/* Rupiah Version */
public class C {
  ThisType tt;
}

public class D extends C {
}
```

```
/* Trsanslated Version */
public interface $$CIfc {}
public class C implements $$CIfc {
  protected PolyClass $$ThisType;
  public C() {
    super();
    $$ThisType = new PolyClass($$CIfc.class);
  }

public interface $$DIfc extends $$CIfc {}
public class D implements $$DIfc {
  public D() {
    super(); // calls C()
    $$ThisType = new PolyClass($$DIfc.class);
  }
}
```

Figure C.2: Initializing `$$ThisType` fields in a class

```
ltarray[0] = new List<T>(null, null);
```

translates to:

```
ltarray.data[0] = new List(null, null, T$$class);
```

If we access an array element, we almost always have to insert a cast because the `data` instance variable has static type `Object[]`. However, `data` will always be an array of the correct type, so this cast never fails.

```
List<T> myList = new List<T>(null, ltarray[0]);
```

is translated to

```
List myList = new List(null, ((List[])ltarray).data[0], T$$class);
```

In this example, we insert a cast to `List[]` because `ltarray.data` has declared type `Object[]`.

Because the `length` field and `clone` methods are already defined in `RupiahArray`, they do not require any additional translation. However, the `Class` object for an array with an element type that is a "`Rupiah` type" is not correct. Because arrays are wrapped up in `RupiahArray` classes, the `Class` object won't be an array type, the `Class` representing `RupiahArray`. Further, the element type is not accurately represented in the `Class`

```
//Rupiah code
object instanceof List<T>[]

//translation
(($synth_0$ = object) == null
  ||
(($synth_0$ instanceof RupiahArray)
  && ((RupiahArray)$synth_0$).$instanceOf$(new PolyClass(List.class,
                                new PolyClass[] {T$$class})))
```

Figure C.3: Translation of `instanceof` for `Rupiah` arrays

object. However, this change only causes a problem for a programmer who makes use of reflection (full reflection is not currently supported in `Rupiah`). Notably, the fact that all `Rupiah` arrays are translated to the same type (`RupiahArray`) does not cause a problem for the `Rupiah` compiler. It maintains their declared type internally and all type checking is performed based on the declared type, not `RupiahArray`.

C.2 `instanceof` expressions

By storing extra type information with arrays, we can translate `instanceof` and casts involving `Rupiah` types as follows. For `instanceof`, we check that the object is an instance of `RupiahArray`, and that the element types are equivalent. Casts are translated similarly, and throw a `ClassCastException` if the run-time validity check fails. An example of the translation for `instanceof` is shown in Figure C.3 and for casts in Figure C.4. As with the other translations for these operations, synthetic variables are inserted as necessary.

Casts and `instanceof` expressions for arrays of type `T`, a type parameter, or `ThisType` are translated similarly. Recall that a representation of the run-time type of `T` is stored in the instance variable `T$$class`. Thus, to check if an object is an `instanceof T[]`, we pass `T$$class` to the `$instanceOf$` method as illustrated in Figure C.5. Similarly, to check if an object is an `instanceof ThisType[]`, we use the `$$ThisType` instance variable as shown in Figure C.6.

C.3 Arrays as instantiations

Array types may also be used as instantiations of type variables. Thus, we must be able to handle operations such as `instanceof` and casts when a type variable `T` is instantiated with an array type. That is, if a type variable `T` is instantiated with `Integer[]` then an

```
//Rupiah code
(List<T>[])object

//translation
($synth_0$ = object) == null
   ||
(($synth_0$ instanceof RupiahArray)
  && ((RupiahArray)$synth_0$).$instanceOf$(new
     PolyClass(List.class,
               new PolyClass[]
               {T$$class})))
 ? (RupiahArray)object
 : throw new ClassCastException()
```

Figure C.4: Translation of casts for `Rupiah` arrays

```
//Rupiah code
object instanceof T[]

//translation
($synth_0$ = object) == null
  ||
(($synth_0$ instanceof RupiahArray)
  && ((RupiahArray)$synth_0$).$instanceOf$(T$$class))
```

Figure C.5: Translation of `instanceof T[]`

```
//Rupiah code
new ThisType[5]
object instanceof ThisType[]

//translation
new RupiahArray(
   java.reflect.Array.newInstance(T$$class.getBase(),
                                  new int[] { 5 }), $$ThisType)
($synth_0$ = object) == null
  ||
(($synth_0$ instanceof RupiahArray)
  && ((RupiahArray)$synth_0$).$instanceOf$($$ThisType))
```

Figure C.6: Translation of `instanceof ThisType[]`

object of type `T` is an instance of `Integer[]`. When we create the `PolyClass` object for an array type as an instantiation, we store the `Class` object representing an array of the base type in the `base` field of the `PolyClass` object. Therefore, our translation for

```
List<Integer[]> lia = new List<Integer[]>();
```

is

```
List lia = new List(new PolyClass(Integer[].class));
```

Then the existing translation for `instanceof T`, where `T` is a type variable, that is based on reflection works correctly because the fact that `T` is actually instantiated with an array type is represented in the `base` member of `T$$class`.

However, if an array whose element type is a type variable is used as an instantiation, then a more complicated translation is needed. In order to help with this translation, a helper method, `makeClassArray`, in `RupiahArray` is provided to obtain `Class` objects representing array types given a `Class` object representing the element types. For example, to create the array `T[][][]`, where `T` is a type variable, we require the `Class` object representing a three dimensional array of `T`'s base type. We write:

```
makeArrayClass(T$$class.getBase(), 3)
```

The `makeArrayClass` method is shown in Figure C.7.

To translate instantiations of type parameters with arrays of type variables, we create a new `PolyClass` object where `base` is the `Class` object representing an array of `T`'s base type. Then we reuse `T`'s existing parameters for the `PolyClass` representing `T[]`. For example, the expression

```
//given a Class c, return Class object representing
//c[]...[] (where number of dims is specified by dims)
public static Class makeArrayClass(Class c, int dims) {
  if (dims <=0 ) {
     return c;
  }
  else {
    Class c1 = makeArrayClass(c, dims -1);
    return java.lang.reflect.Array.newInstance(c1, 0).getClass();
  }
}
```

Figure C.7: `RupiahArray makeArrayClass` method

```
List<T[]> lta = new List<T[]>();
```

translates as

```
List lta =
  new List(new PolyClass(makeArrayClass(T$$class, 1),
                         T$$class.getParams()));
```

Instantiations of type variables with arrays of type `ThisType` are translated similarly, using the same `$$ThisType` instance variable as in Figure C.6.

C.4 Multi-dimensional arrays

Multi-dimensional arrays of `Rupiah` types are translated as arrays of `RupiahArray`. Only the final dimension is created with the actual element type. To help with this translation, we provide a static helper method in `RupiahArray` named `multiArray` that handles the creation of multi-dimensional arrays, shown in Figure C.8.
The two-dimensional new array expression:

```
new List<T>[5][4]
```

becomes

```
RupiahArray.multiArray(new PolyClass(
        List.class, new PolyClass[] { T$$class }),
        2, new int[] { 5, 4 })
```

```
public static RupiahArray multiArray(PolyClass type,
                      int dims, int[] dimSizes) {
  if (dims == 0)
    return new RupiahArray(
      java.lang.reflect.Array.newInstance(type.getBase(),
                           new dimSizes[dimSizes.length]),
      type);
  }
  else {
    RupiahArray data =
      new RupiahArray(new RupiahArray[dimSizes[dimSizes.length - dims]],
                      new PolyClass(makeArrayClass(type.getBase(), dims),
                                    type.getParams()));
    for(int i = 0; i < data.length; i++) {
      data[i] = multiArray(type, dims - 1, dimSizes);
    }
    return data;
  }
}
```

Figure C.8: `RupiahArray multiArray` method

Each dimension except for the final one is a `RupiahArray[]`. This allows us to correctly execute casts and `instanceof` expressions for multi-dimensional arrays of `Rupiah` types. Thus, we can translate expressions such as

```
object instanceof List<Integer>[][]
```

in a straight forward manner:

```
Object $synth_0$;
((($synth_0$ = object) == null)
 ||
(($synth_0$ instanceof RupiahArray)
  && ($synth_0$.instanceOf(new PolyClass(List[].class,
                           new PolyClass[] {Integer.class})))))
```

Casts to multi-dimensional array types are translated similarly.

C.5 Static Array Initializer Expressions

In Java, one may declare and initialize a new array in a single expression. For example, the expression

```
new int[] { 1, 2, 3, 4}
```

initializes an integer array of length 4 with values `1, 2, 3, 4`. One dimensional new static array initializers like this one are easy to translate for most types. For example, the expression

```
new List<T> { null, null, null }
```

translates to

```
new RupiahArray(new List[] { null, null, null }, T$$class)
```

If the element type is a type variable, then the situation is more difficult. Because the element type is not known statically, we introduce another helper method in `RupiahArray` to perform the actual initialization. Reflection is used to create the array with the correct element type (stored in a `PolyClass` object), then the supplied initial values are copied into this array.

```
public static RupiahArray staticArray(Object[] data, PolyClass type) {
  Object[] newData =
     java.lang.reflect.Array.newInstance(type.getBase(), data.length);
  System.arraycopy(data, 0, newData, 0, data.length);
  return new RupiahArray(newData, type);
}
```

Thus the expression

```
new T[] { null, null}
```

becomes

```
staticArray(new Object[] { null, null}, T$$class);
```

Multi-dimensional static array initializers can be reduced to nested single dimension static array initializers. For example, the expression

```
new T[][] { new T[] { null }, new T[] { null}}
```

translates as

```
staticArray(new Object[] { staticArray (new Object[] { null }, T$$class),
                           staticArray (new Object[] { null }, T$$class)},
            T$$class)
```

C.6 Type Rules

A consequence of this translation is that the Java rule that allows array types to be subtypes if their element types are subtypes is not allowed for `Rupiah` programs. Array types are only subtypes if their element types are identical. This fixes a well known hole in the static type system, illustrated by the following code:

```
//D is a subtype of C
C[] cArray = new D[10];
cArray[0] = new C();    //throws ArrayStoreException
```

However, it means that the operational semantics for some code changes in `Rupiah`. Consider the following code, compiled with a standard Java compiler:

```
void foo(Object o) {
  if (o instanceof Object[])
     //side-effect
}
```

`foo` should execute its side-effect for any `o` which is an array of objects. However because arrays made up of `Rupiah` types are wrapped in `RupiahArray` objects, the `instanceof` test fails for any `RupiahArray` that is passed as the actual value of `o`.

Therefore, for consistency we eliminate the covariant subtyping rule for array types. No expressiveness is lost by making this change: we have a generic types so any code that was written making use of this rule can be written instead with with `T[]`, where `T` is a type

variable, and eliminate the need both this unsafe rule and any run-time checks. Thus, we also don't need to worry about throwing an `ArrayStoreException` (we'll never have to for code that passes the type checker). However, we lose compatibility with pre-compiled Java code which makes use of this unsafe rule.

If the programmer must absolutely use the covariant array subtyping rule, they may pass an `-unsafe-arrays` flag to the compiler which means that for non-`Rupiah` type arrays, the standard Java rule is used. However, any `Rupiah` array is still not a subtype of `Object[]`. Also, `Rupiah` arrays are never subtypes unless their element types are identical, even with the `-unsafe-arrays` flag.

C.7 Evaluating the Array Proposal

This solution, which involves boxing and unboxing, is not ideal. An obvious drawback is the overhead of storing each array inside of another object. Further, as most accesses to an array require a cast, we introduce new dynamic checks in our translation of arrays. However, efficiency problems are not our foremost concern. Instead, the fact that this translation breaks compatibility with existing Java code is the more troubling issue. We would like to be able to compile any existing Java source code, and call any pre-compiled code from `Rupiah` programs. However, our translation of arrays in `Rupiah` forces us to eliminate the rule that array types are subtypes if their element types are subtypes. This does not make our language any less expressive: a class `Array<T>` can easily be written using parametric polymorphism. However, it does increase the distance between the `Rupiah` programming language and Java. Unfortunately, we see no other practical solution that both allows array types in `Rupiah` to be valid in operations like casts and `instanceof` and does not require changes to the JVM. Ultimately, the problem of properly supporting arrays in `Rupiah` may be the factor which drives the project towards a translation based in part on a modified JVM.

Appendix D

Compiler Information

Our distribution of the `Rupiah` compiler is available electronically. Latest documentation, binaries and source code are linked from the project web page. The `Rupiah` compiler itself is based on Sun Microsystems `javac` compiler. The version of `javac` used in project was 1.2.2. The binary distribution of `Rupiah` is the `rupiah.jar` file. The compiler can be executed with the command:

```
java -jar rupiah.jar [options] <source files>
```

or alternatively

```
java EDU.williams.tools.javac.Main [options] <source files>
```

As the `Rupiah` compiler is based on `javac`, the command line options are identical. The set of classes (`PolyClass`, `$$MangleClass`, etc.) that are needed to run `Rupiah` programs are also distributed as a separate archive, `rupiahrt.jar`.

The URL for the `Rupiah` project webpage is:

```
http://www.cs.williams.edu/~kim/rupiah/
```