The Design and Implementation of the Tako Language and Compiler

Jyotindra K. Vasudeo

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Dr. Gregory Kulczycki, Chair
Dr. William B Frakes
Dr. Ing Ray Chen

May 5th, 2006
Falls Church, Virginia, USA

Keywords: Aliasing, Language Design, Compilers, Formal Reasoning, Java, Tako

The Design and Implementation of the Tako Language and Compiler

by Jyotindra Vasudeo

Abstract

Aliasing complicates both formal and informal reasoning and is a particular problem in object-oriented languages, where variables denote references to objects rather than object values. Researchers have proposed various approaches to the aliasing problem in object-oriented languages, but all use reference semantics to reason about programs. This thesis describes the design and implementation of Tako—a Java-like language that facilitates value semantics by incorporating alias-avoidance. The thesis describes a non-trivial application developed in the Tako language and discusses some of the object-oriented programming paradigm shifts involved in translating that application from Java to Tako. It introduces a proof rule for procedure calls that uses value semantics and accounts for both repeated arguments and subtyping.

Table of Contents

List of figures

Acknowledgments

I would like to thank my advisor, Gregory Kulczycki, for his support, care, and patience during my time as a graduate student. His insight and ideas helped form the foundation of this thesis. His guidance and support has transformed a struggling graduate student into a researcher. He has taught me how to perform scholarly research, think creatively and independently, and strive for excellence. I only hope that after graduating, I can practice what I have learned from him. I would also like to express my gratitude to Bill Frakes and Ing-Ray Chen for being on my thesis committee.

I dedicate this work to my family, to whom I owe the greatest debt. I would like to thank them for firmly believing in me and constantly encouraging me during easy and uneasy times. I could not have come this far without their love, support, and sacrifices. I am grateful to them for their selfless decisions, practical advice, tireless encouragement, and unconditional support.

Chapter 1

Introduction

Aliasing complicates both formal and informal reasoning [Hoa 75] because "changes to an object potentially affect all objects that refer to it, even though the object being changed may be unaware of the other objects" [Cla 98]. This is especially true in object-oriented languages, where variables denote references to objects rather than object values [Hog 92]. Researchers have proposed various solutions to the problem of object aliasing. These approaches can be divided into two broad categories: alias control and alias avoidance. Alias-control techniques [Nob 98, Cla 98, Alm 97] focus on limiting the effects of aliasing while preserving traditional object-oriented paradigms. Alias avoidance techniques focus on preventing common sources of aliasing. They typically involve replacing reference assignment with non-traditional alternatives, such as destructive read [Min 96] and swapping [Har 91].

Alias-avoidance techniques are less common than alias control techniques because they increase the risk of upsetting established OO paradigms. Alias-avoidance based on swapping is less common than alias-avoidance based on destructive read for similar reasons.

This thesis describes the Tako project, whose main objective is to simplify object-based reasoning by introducing alias-avoidance into a popular object-oriented language. Specifically, it is concerned with redesigning the Java language with alias avoidance techniques based on the swapping paradigm [Har 91, Wei 02]. We chose Java for at least two reasons. First, the language is popular and its popularity continues to grow. This allows us to illustrate the paradigm shifts involved in using alias avoidance to the largest audience possible. Second, we had access to an open source Java compiler that we could easily modify. We chose the swapping approach because of our familiarity with it and because of the lack of research on directly integrating it into object-oriented languages.

In pure functional languages, side-effects in functions are disallowed. This provides referential transparency similar to the alias-avoidance methodology as described above. In this thesis we are only concerned with imperative languages. More specifically we only deal with alias-avoidance concepts in object-oriented languages like Java. Also, we do not consider domain specific languages; instead we focus on general purpose programming languages that can be used for all programming purposes.

The two main research questions that we hoped to answer with the development of the compiler were (1) Can alias-avoidance techniques be incorporated into an object-oriented language in a way that preserves object-oriented features but also supports value semantics? and (2) Can Java programmers adjust to the paradigm shifts involved in using alias avoidance techniques based on the swapping paradigm? Our experience with the Tako project gives us reason to be optimistic on both questions.

The thesis makes the following contributions to research in object aliasing:

- It describes the implementation of the Tako compiler, including its scheme for in-out parameter passing, automatic initialization, and handling of repeated arguments.

- It describes a non-trivial application developed in the Tako language and discusses some of the paradigm shifts involved in translating that application from Java to Tako.

- It introduces a proof rule for procedure calls that uses value semantics and accounts for both repeated arguments and subtyping.

The development of the compiler has several benefits:

- It allows us to study the paradigm shifts involved in moving from Java to Tako.

- It allows us to understand what aspects of object-oriented languages would have to change to incorporate alias-avoidance techniques.

- It gives us a tool that can serve as the basis for a formal verification system (verifying compiler).

Work on the Tako project started in late 2004. We registered the Tako compiler as an open source project on SourceForge and released version 1.0 of the compiler in early 2006. Also, we set up a wiki page for the project at http://www.directreasoning.org/wikitako to allow discussions between the participants of the project and also to document our findings during the implementation of the compiler. The development IDE (Integrated development environment) used for the project was Eclipse.

Several times, the compiler development directly influenced our research into language design and alias avoidance. For example, we were not aware of the potential problems involved with our version of in-out parameter passing and Java-style inheritance until we tried to implement pseudo-generic types in Tako. Also, the distinguished "result" variable was not included in the language until we noticed aliasing occurring when certain functions returned.

The outline of this thesis is as follows. Chapter 2 talks about the related work done by other researchers. Specifically, it talks about the existing research related to aliasing and formal verification. Chapter 3 discusses the features of the Tako language in detail. Chapter 4 describes the implementation of the Tako compiler in Java. Chapter 5 gives proof rules for common statements in the Tako language. It also introduces a new rule for procedure calls that can handle both repeated arguments and Java-style subtyping and inheritance. Chapter 6 presents a brief case study that illustrates the paradigm shifts involved when moving from Java to Tako. Chapter 7 contains our conclusions and future work.

Finally, Appendix A gives example proofs using proof rules given in Chapter 5 and Appendix B gives the Java translation as generated by the compiler for two example Tako classes.

Chapter 2

Related Work

This section discusses existing research on object aliasing and its impact on reasoning. Before the popularity of object-oriented languages, Hoare [Hoa 75] and Kieburtz [Kie 76] noted that references and aliasing in programming languages complicated reasoning. As object-oriented languages became popular, the problems related to aliasing grew more prevalent as most complex type variables where implemented as references. A key paper titled *The Geneva Convention on the Treatment of Object Aliasing* [Hog 92] discussed these issues in detail. It mentioned four ways to tackle the problems related to aliasing: alias detection, alias advertisement, alias avoidance, and alias control. Alias detection is essentially a post hoc activity where aliasing is determined in a program through static or dynamic analysis. The static detection may require NP-hard analysis which may not be acceptable. Alias advertisement requires annotating methods to advertise their aliasing properties but does not specifically enforce them. While these methods have their place, we focus here on alias control and alias avoidance.

2.1 Alias Control

Alias control techniques attempt to limit the effect of aliasing. For example, balloon types [Alm 97] strengthen the concept of encapsulation by avoiding any aliasing to *balloon* objects, which are indicated using the *balloon* keyword during type declaration. The compiler that restricts aliasing into balloon objects statically checks that programmers do not violate the invariants. Other techniques such as Islands [Hog 91] share similarities with Almeida's balloon types.

Flexible alias protection provides special keywords and syntax such as *rep* and *arg R*, where *R* is the specified role, used with variables, types and methods. The keyword indicates to the compiler what alias restriction is desired. Thus the burden is transferred to the programmer in terms of additional syntax, making static checking easier. This also allows for more flexibility and thus a more natural object-oriented style of programming than Balloon types or Islands. Clarke has adapted flexible alias protection by introducing the additional concept of owners for objects [Cla 98].

In these techniques, special annotations (similar to access modes) are added to the language and the compiler performs static checks to enforce alias control. The annotations complicate the language but give the programmer the flexibility to revert to a traditional object-oriented style of programming. The semantics for these languages are reference based rather than value based.

2.2 Alias Avoidance

Alias avoidance techniques prevent common sources of aliasing. One such technique avoids pointer aliasing by using the concept of *unique pointers* and *unshareable objects* [Min 96]. An unsharable object or u-object is only referenced by a single pointer variable and this pointer is called the u-pointer. Obviously normal pointer copy operations cannot be used with u-pointers; instead a move operator is used. The move operator copies the pointer address from the left hand side variable to the right hand side and then nullifies the left hand side variable. This operation is more commonly known as a destructive read operation.

The RESOLVE research has also extensively used alias avoidance techniques. RESOLVE is an overloaded term that can represent a framework, language, and/or a general discipline for using a language [Sit 94]. RESOLVE research focuses on component-based software and uses the swapping paradigm to help avoid aliasing. [Hol 00] describes how the Resolve discipline used in conjunction with C++ was successfully used for development of a commercial application. Many Tako features are derived from RESOLVE-style alias avoidance.

RESOLVE as a language is not object-oriented. One of the key questions we had at the beginning of this project is how RESOLVE-style alias avoidance would work with Java style object-oriented features.

2.3 Verification

With Tako, we also hope to explore value based verification systems for object-oriented languages. This sub-section looks at some of the current verification research in object-oriented languages.

2.3.1 Runtime Assertion Checking

The Java Modelling Language (JML) is a behavioral specification language for Java. JML specifications are written in a functional subset of Java, and JML is intended to support both runtime assertion checking and formal verification [Lea 02]. The specifications can include model variables, abstraction functions, invariants, and basic requires and ensures assertions. The JML tool can check many (but not all) of these assertions at runtime.

2.3.2 ESC/Java

The Extended Static Checker for Java (ESC/Java) [Lei 00] can statically check Java code for common errors such as the possible occurrence of null pointer exceptions. It also understands a subset of JML and can statically verify some lightweight specifications.

2.3.3 LOOP compiler

The LOOP compiler [Van 01] is intended for full verification of sequential Java programs. To prove the correctness of the Java code with respect to its JML specification, the compiler translates both the code and the specification into a heap-based logic. The resulting logical theories are then proved with the help of the PVS theorem prover.

2.3.4 RESOLVE

The RESOLVE verifying compiler focuses on modular, heavyweight verification using a value-based semantics [Sit 94]. No heap structures or special proof rules for pointers are needed. The state space at any point in the program is comprised of the programming and conceptual variables and their values. The frame property restricts the effects of an operation invocation to its actual parameters and the global variables declared in the updates clause of the operation declaration.

Ideally, we would like to be able to implement runtime assertion checking for Tako as well as automatic static checking of lightweight specifications in the spirit of ESC/Java. For full verification, however, we aim for a RESOLVE-style system with its simpler value semantics, state space, and frame property. In particular, we would like to investigate about how to reason about Java-style inheritance using the simple frame property given in RESOLVE-style semantics. Thus, the Tako project aims towards building a basis for an object-oriented verifying compiler using value-semantics.

Chapter 3

Features

Alias avoidance is the distinguishing feature between Tako and Java. In Java, variables denote references to objects, making aliasing routine; in Tako, variables denote objects, making aliasing rare. Java supports reference semantics, while Tako supports value semantics. A key goal of Tako is to support value-based semantics while easing the paradigm shift for object-oriented (in particular Java) programmers. Thus, Tako can be thought of as a redesign of Java that incorporates alias-avoidance techniques. The redesign includes features such as automatic initialization, alternative data assignment operators, in-out parameter passing and a pointer component. This section discusses these features in detail.

3.1 Automatic Initialization

When variables are declared in Java, they do not reference any object until assigned to by a constructor or some other variable. In Tako, which supports a view of variables as values, variables are initialized with a default value of their type as soon as they are declared. This scheme is consistent with the primitive types of Java, which are initialized when they are declared. Thus, the declaration *Stack s;* in Tako is treated as being operationally equivalent to *Stack s := new Stack();*.

Even though default values are generated for all declared variables in Tako, programmers must not depend on this feature. For example, if a programmer creates a bounded stack, the default stack object will have a bound of zero, which is effectively unusable. In this case, it is better for programmers to explicitly assign a new *Stack* object to a newly declared variable, as in *Stack s := new Stack(20);* In general, programmers are encouraged to provide their own constructors for all types, similar to Java.

3.2 Data Assignment

Since reference assignment is the main cause of aliasing in Java, Tako avoids it. Instead, Tako uses alternatives such as swapping and initializing transfer. The Figure 3.1 shows how various forms of data assignment affect the variables and the objects they represent.

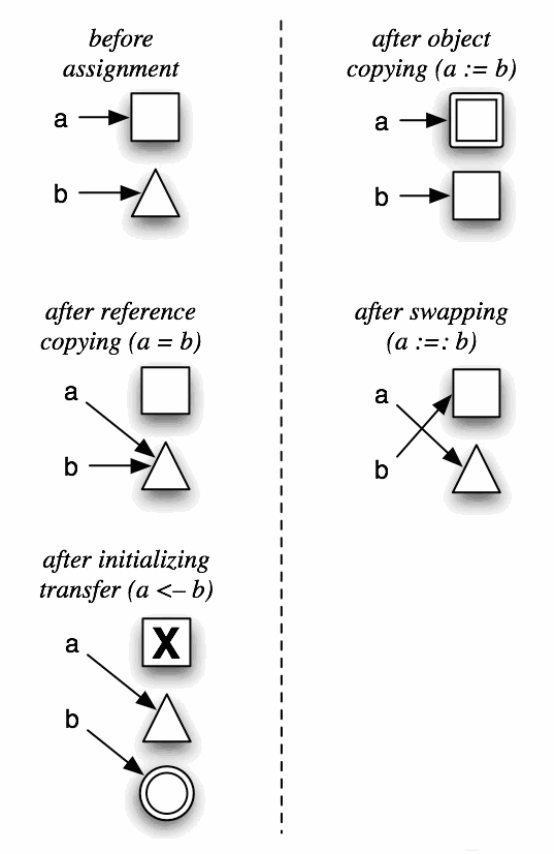

Figure 3.1 The effects of various forms of data assignment

3.2.1 Swapping

Swapping has been proposed as an alternative for reference assignment and value copying [Har 91]. Swapping can be implemented by swapping of object references, resulting in a

constant time operation as shown in Figure 3.1. Many machine languages already implement the swap operation as an atomic instruction, thus making it equivalent in performance to the reference assignment operator. Since it is efficient and avoids aliasing it is the primary means of data assignment in Tako. In Tako the swap operator is denoted by the symbol `:=:` .

A criticism of swapping is that it is a symmetric operation, and is not meant to swap variables of different types [Cla 98]. For example, programmers may use the reference assignment operator to assign a subtype object to a supertype as in *s = c;*, where *s* is of type *Shape* and *c* is of type *Circle*. A similar operation *s* `:=:` *c* cannot be performed with the swap operator because variable *c* will point to an object of type *Shape*, which is a runtime violation. Thus, for the swap operation to be permitted both variables involved should be of the same static type.

To overcome this limitation, the initializing transfer operation is used.

3.2.2 Initializing transfer

The initializing transfer operator can be used in asymmetric operation as shown in Figure 3.1. It is denoted by the symbol `<-` in Tako. In the statement *a* `<-` *b;* the object reference of *b* is copied to *a*, and *b* gets an initial value. It is clear that the operation is valid even if *a* and *b* have a subtype relationship. Similar to the swapping operation, the initializing transfer operator does not cause aliasing.

Initializing transfer, unlike swapping, involves initializing the right hand side variable. This can be costly compared to reference assignment and swapping. A lazy initialization scheme could improve the performance.

3.2.3 Function assignment

Methods in Java return references rather than values, potentially introducing aliasing. To avoid this in Tako, return values in methods are stored in an internal *result* variable. The *result* variable is initialized at the beginning of the function and whatever value it holds at the end of the

method is returned. This ensures that a unique reference is returned. In Tako methods an explict *return* statement is not needed as the *result* variable is implicitly returned at the end of the function.

To assign this return value from methods to variables the function assignment operator is used. In Tako it is denoted by the symbol :=. Apart for assigning the return values, the function assignment operation is also used for copying objects or primitives. The statement *s := t;* is syntactically valid in Tako. At first glance, it may seem to be similar to a reference copy operation, in fact it is equivalent to *s := t.replica();*, where *replica()* is a user defined function which makes a copy of type of *t*. Similarly if *a* and *b* are of primitive types, the function assignment *a := b* can be used to copy the value of *b* to *a*.

3.3 Parameter passing

In Java, parameters are passed by copying the references of the actual parameters into the formal parameters, the references are not copied back from the formals to the actuals after the call. Conceptually, this makes it difficult to reason in a language that facilitates value-semantics. Firstly, reasoning cannot be done without introducing the notion of references. Secondly, only copying references in does not always allow programmers to update the argument's conceptual object value. To avoid this problem, Tako supports in-out parameter passing for parameters.

In Tako, parameter passing can be viewed as transferring the actual parameters, to the formals via initializing transfer, executing the method, and then transferring the formal parameters back to the actual parameters. This view of in-out parameter passing avoids aliasing in the case when the arguments repeated as parameters, as in the call *q.append(q);*. A variation of this approach to parameter passing also avoids type errors when Java-style polymorphism and subtyping is used. These issues are examined in formal proof rules for procedure calls in detail in Section 5.3, where we discuss formal proof rules for procedure calls.

3.4 Pointer Component

The main objective of Tako is to simplify the reasoning of code written in object-oriented languages by alias avoidance. In most cases aliasing can be avoided and alternative data assignment operators such as swapping, can be used to construct classes. But in certain circumstances the use of references cannot be avoided for efficiency sake. For example, to construct a *LinkedList* class, references are needed to ensure that the fundamental operations on the list run in constant time. In order to implement such linked data structures Tako provides a pointer component. This pointer component is based on the reasoning approach for pointers described in [Kul 05a].

The interface for the pointer component is in the form of a Position interface and is shown in in Figure 3.2.

```
interface Position {
  static final int k;
  public void takeNew();
  public void moveTo(Position p);
  public void redirectLink(int k, Position p);
  public void followLink(int k);
  public void swapContents(Object x);
  public boolean isWith(Position p);
  public boolean isAtVoid();
}
```

Figure 3.2 The Position Interface class

This view of pointers as Position objects helps the programmer to reason about them like any other object. It also adheres to the value-based reasoning system of Tako. No special proof rules are needed for pointers, and no universal heap structure needs to be included in the semantics.

Despite the fact that pointers can be viewed as objects in Tako, implementing pointers as objects would be inefficient. Therefore, pointers have special implementation which is discussed in detail in Section 4.3.4.

Chapter 4

Implementation

Our desire to keep the Tako language syntactically similar to Java lead to the Tako compiler being built on top of an existing Java compiler - the Kopi compiler. The Kopi compiler is an open source compiler developed in Java that compiles Java source code to byte code. The available version of the Kopi Compiler (version 2.1B) supports Java 1.4 syntax and includes support for Java style assertions. The compiler also has support for custom style generics that differ slightly from Java 1.5 style generics. The Tako compiler is built on top of the Kopi compiler, which made support for Java 1.4 syntax and semantics easier to implement. To support Tako syntax only slight modifications were needed to the Kopi compiler due to similarity between Tako and Java syntax. Tako 1.0 code which is similar in appearance to Java 1.4, but the byte code generated is Java 1.5 compatible. This is because we found it beneficial to use Java 1.5 features (such as annotations) during translation from Tako to Java. Since Java 1.4 has no support for generics, the custom implementation of generics in the Kopi compiler was disabled. Later versions of Tako maybe similar to Java 1.5 in syntax and have support for generics also. The present version of the Tako compiler does not support assertions as intended, this is because of issues inherited from the assertion implementation from the Kopi compiler.

Let's explore in some detail the Kopi compiler design so as to better understand the Tako compiler.

4.1 Kopi Compiler Design

The Kopi compiler has roughly four stages of operation: parsing, pre-analysis, analysis, and byte code generation as shown in the Figure 4.1. Errors generated in any of theses stages are given to the error output component which prints the formatted error message to the user.

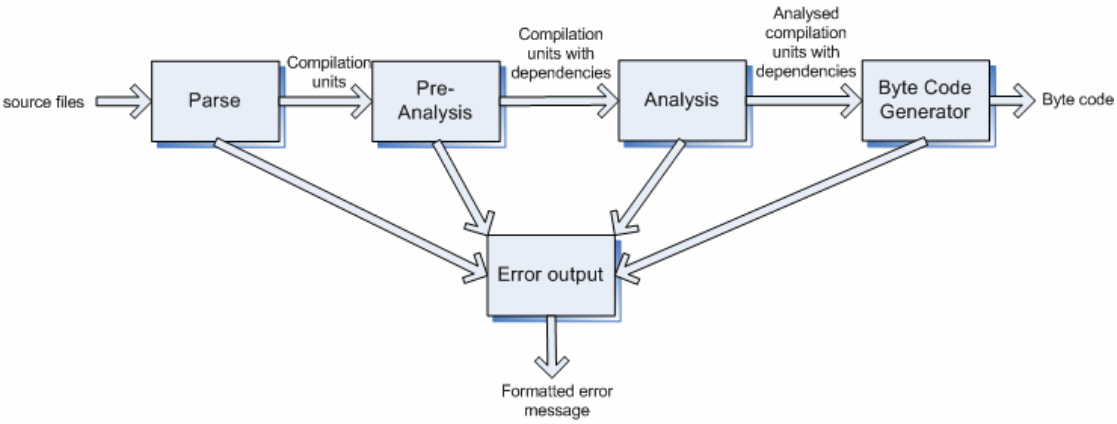

Figure 4.1 The Kopi compiler design

The parsing of the source files is done with a parser created by the ANTLR parser generater. The source files are parsed and compilation units are generated for each file. The compilation units represent the nodes of an Abstract Syntax Trees (AST) for a given source file. These ASTs can then be traversed to perform appropriate analysis and code generation. Parsing may produce syntax errors if the code does not conform to the grammar.

Initially, dependency information such as location of the imported classes and interfaces is absent in the compilation units. This information is added to the compilation units during the pre-analysis stage. If the dependent classes are missing then appropriate errors are generated during pre-analysis.

The analysis stage checks for semantic errors that may be present in the code. The Java Language Specifications (JLS) for Java 1.4 [Gos 05] is used for this purpose. During analysis, the abstract syntax tree is traversed and an analyze method for each node is executed to check

for these errors. Usually, the analyze method of a node is invoked from within the analyze method of its parent. Using this hierarchy of calls, each node of the AST can be visited and analyzed. The anlayse method of the root is invoked to start this process. The visitor design pattern [Gam 94] could easily have been used instead of the above technique. The visitor design pattern separates code with complex node hierarchies into data (nodes) and functionality (visitors). It is used when the data does not often change, but functionality needs to be easily extended. This would have obvious advantages for analysis. All the analysis code could be isolated to only one file and not spread across numerous nodes. More than one visitor could be employed, thus further separating code based on functionality. Changes to the analysis code could easily be made due to this segregation of code. The designers of the Kopi compiler implemented the visitor pattern, but did not make use of it for analysis. Why they did this is unclear, considering the advantages.

The analyzed compilation units generated from analysis are then fed to the byte code generators which produce the *.class files.

4.2 Redesign of the Kopi compiler for Tako

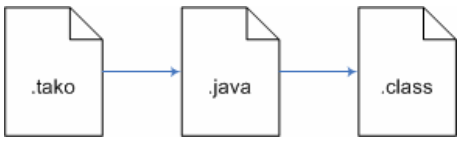

Figure 4.2 Tako source to bytecode

The Tako compiler has to translate Tako source (*.tako) files to Java source (*.java) files, which are then converted to byte code (*.class) files. To accommodate this in the Kopi compiler the following changes were made.

The grammar file was tweaked to accept Tako 1.0 syntax. The Tako 1.0 syntax is very similar to Java 1.4 syntax, except for a few data assignment operators and certain keywords. The compiler was made to accept *.tako files rather than *.java files.

Pre-analysis and post-analysis visitors were added to perform translation and analysis tasks. Unlike Kopi analysis we decided to use the framework of the existing visitor pattern to do our analysis. All the Features of Tako 1.0 are implemented in the form of visitors which traverse the abstract syntax trees and make changes to it as needed. Some features which require direct textual substitutions such as swapping and initializing transfer are implemented as pre-analysis visitors. But features such as in-out parameter passing, and automatic initialization which need type information are implemented as visitors after the analysis stage. Changes made to any section of the AST after analysis are re-analyzed to ensure that the type information remains up-to-date.

Once all analysis is completed, instead of passing the modified AST through the Kopi byte code generator, it is passed through a java pretty printer which generates Java 1.5 source files. These source files are now potentially free of any kind of syntactic or semantic errors. They are then fed to the Java 1.5 compiler (javac) which produces the final class files. The Kopi byte code generator could not be used for this purpose because after analysis, the AST is Java 1.5 code complaint, but the byte code generator is only meant for Java 1.4 code. Thus, we had to use Java's own compiler to perform this function.

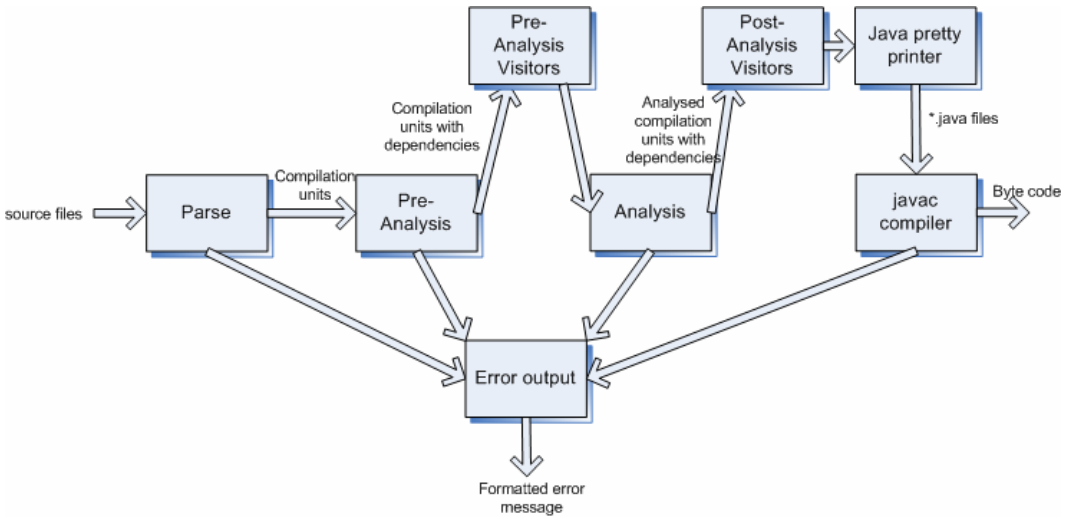

Figure 4.3 Tako compiler design

4.3 Feature implementations

Tako version of *Stack*

```
public class Stack {
  private Object[] contents;
  private int top;
  private final int MAX;

  public Stack(int size) {
    MAX := n;
    contents := new Object[MAX];
    top := -1;
  }

  public void push(Object x) {
    assert depth() < MAX;
    top++;
    contents[top] :=: x;
  }

  public void pop(Object x) {
    assert depth() > 0;
    x <- contents[top];
    top--;
  }

  public static void main(String[] args)
  {
    Stack s;
    s = new Stack(20);

    /* rest of main */
  }
}
```

Java translation of *Stack*

```
public class Stack {
  private Object[] contents =
                              new Object[0];
  private int top;
  private final int MAX;

  public Stack(int size) {
    MAX = n;
    contents = new Object[MAX];
    for(int i=0; i<MAX; i++) {
      contents[i] = new Object();
    }
    top = -1;
  }

  public void push(Object x) {
    assert depth() < MAX;
    top++;
    Object temp = contents[top];
    contents[top] = x;
    x = temp;
  }

  public void pop(Object x) {
    assert depth() > 0;
    x = contents[top];
    contents[top] = new Object();
    top--;
  }

  public static void main(String[] args)
  {
    Stack s = new Stack();
    s = new Stack(20);

    /* rest of main */
  }
}
```

Figure 4.4 Stack implementation in Tako (left) and Java translation (right)

As discussed in the last section, most features of Tako are implemented in the form a visitor which traverses the tree either before or after the analysis stage. This allowed for better maintenance and segregation of the code for each feature. Although all the individual visitors could be integrated into a single visitor for optimization reasons, the code could easily become unmanageable in such a scenario. Therefore we decided to adhere to one (or more) visitor per feature.

The next few subsections look at the implementation of each feature in detail.

4.3.1 Automatic Initialization

The compiler supports the automatic initialization of the following variable types

- Primitive Types
- Classes
- Interfaces
- Abstract classes, and
- Arrays

Similar to Java, primitive types in Tako are initialized using their default values. Thus, the compiler doesn't need to do anything special in terms of translation to Java and can keep the same Tako code for primitive type variable declarations. In Figure 4.4, the variable declaration ***private int*** *top;* in Tako remains unchanged in the Java translation. The compiler need not bother with initializing top with default value of zero, as this will be automatically done by Java.

Java has no feature for automatically initializing object variables and expects the users to perform this task. This is not the case in Tako. To implement this feature in Tako, the compiler needs to add initialization code wherever object variables are declared. This is done using the "newInstance()" method in the "java.lang.Class" class of Java. The "newInstance()" method returns a new instance of the class represented by the "Class" object that corresponds to the variable type. It is equivalent to calling the default (zero argument) constructor of the class. Therefore in the Java translation we can consider the variables being initialized using the default constructor. If an overridden zero argument constructor exists in the class, it is used as the default constructor. If a client wants a non-default value for a variable, he must explicitly assign it. In Figure 4.4. the statement *Stack s;* would be translated to Java as *Stack s = new Stack();*. The member variables of the *s* would get the initial values assigned to it during their declaration when the default constructor is used. In the case of the bounded stack, however, a client will probably want ot use the one argument constructor that gives the stack its bound, as in *Stack s := new Stacl(20);*.

Interfaces and Abstract classes are an exception in Tako. Since they cannot be automatically initialized at compile time, they are assigned the *null* value when declared. These variables can then be assigned some concrete object value as needed. This partially violates the notion of hiding references in Tako, but if the user always remembers to assign some object to the variables before using them, the user will not encounter *null* values.

When arrays are declared in Tako they are initialized as a zero length array. Thus the statement ***private*** *Object[] contents;* in Figure 4.4 is translated to ***private*** *Object[] contents* = ***new*** *Object[0];* in Java. Naturally, the array variable can be explicitly initialized with a non-zero length array, as illustrated by the statement *contents :=* ***new*** *Object[MAX];* in the Figure 4.4. In Tako, unlike Java, each element of the non-zero length array is also automatically initialized by the compiler. However this feature is unimplemented in the present release of the compiler.

The above initialization scheme described though effective has certain drawbacks.

Firstly, this is a very costly implementation of automatic initialization in terms of space and time. If certain variables are not used they still need to be initialized. An alternative implementation would be based on lazy evaluation or lazy initialization. Lazy initialization is similar to lazy evaluation in that variables are allocated space only if they are used. From a reasoning perspective, the programmer can still think of objects as having a default value until it is modified. This optimization is not present in the current version of the compiler.

The other issue is with cyclic classes. If a class contains a member variable of its own type, this may cause an infinite loop due to automatic initialization. This could be avoided if the programmer writes an appropriate default constructor. Such classes are common in Java for implementing "linked" data structures. Tako provides a pointer component for this purpose, which avoids potential initialization problems.

4.3.2 Data Assignment

As discussed, Tako has three data assignment operators: swapping, initializing transfer and function assignment. Unlike Java's reference assignment operator, which not supported in Tako, all these operators avoid aliasing.

Swapping (:=:)

Objects need to be of the same static type before they can be swapped. This ensures that there are no conflicts at runtime when objects are swapped. In the compiler, swapping can be implemented as a simple textual substitution. As seen in the push and pop methods shown in Figure 4.4, the statement *contents[top] :=: x;* is translated to Java as :

```
Object temp = contents[top];
contents[top] = x;
x = temp;
```

The compiler simply swaps the references of *contents[top]* and *x* behind the covers. The present version of the compiler does something similar, however instead of direct textual substitution of the code, a global static swap method in used with the same effect.

Initializing transfer (<-)

With initializing transfer, the variables need not have the same static types, but they should have a subtype relationship. Figure 4.4. shows the use of initializing operator in the pop method in the statement *x* <- *contents[top];*. The Java translation is

```
x = contents[top];
contents[top] = new Object();
```

The variable *contents[top]* is initialized after being assigned to *x*. In this case both *x* and *contents[top]* are of the same type. This operation is also valid with *contents[top]* being a subtype of *x*.

Like swapping, initializing transfer is also currently implemented as a static method instead of inlining the code.

Function Assignment (:=)

The function assignment operator, as it's name suggests, is used to assign the return values of functions to variables. Apart from this, the operator is also used to copy objects and or primitives.

Tako version of *clearTop*	**Java translation of *clearTop***

`public Object clearTop() {`
`  result :=: contents[top];`
`}`
	`public Object clearTop() {`
	`  Object result = new Object();`
	`  Object temp = result;`
	`  result = contents[top];`
	`  contents[top] = temp;`
	`  TKGlobals.result = result;`
	`  return result;`
	`}`

Figure 4.5 clearTop method in Tako (left) and Java translation (right)

Consider the clearTop function shown in Figure 4.5. In the Tako version, the ***result*** keyword is used as a distinguished variable that stores the return value of the function. In the Java

translation, this ***result*** variable is explicitly declared and initialized at the start of the function. The translated method body assigns the value of *contents[top]* to it. Finally, at the end of the function the local result variable is assigned to a global result variable and the value of result is returned. The global result variable can now be used to assign the return value of the function. For example consider the following function call.

```
Object x := s.clearTop();
```

This Tako statement is translated to the following Java code.

```
Object x;
s.clearTop();
x = TKGloabls.result;
```

The function call needs to be decomposed from the assignment statement to avoid assigning an alias of ***result*** to *x*. Once the function is executed, the value of result is stored in the global result variable *TKGlobals.result* which is assigned to *x*. Only *TKGlobals.result* and *x* have a reference to the returned result value. Since *TKGlobals.result* is a compiler variable - used and declared within the compiler - the user's view of an alias free environment remains consistent with this scheme.

Till now we have seen the function assignment operator used with a variable on the left-hand side and a function call on the right-hand side. The operator can also be used with variables on both sides of the operator as discussed. Thus, the following Tako code is also syntactically correct.

```
Stack s := new Stack(20);
Stack t := s;
```

At first glance, the function assignment operator in the above code may seem to behave as a reference assignment operator in Java, but recall that Tako does not have reference assignment. Instead, the above Tako code is semantically equivalent to the following Tako code.

```
Stack s := new Stack(20);
Stack t := s.replica();
```

As noted the user-defined replica method is intended to be used for deep copying of objects. Thus if a *replica()* function is not defined in *Stack*, and error will be generated by the compiler. Any class which is considered replicable in Tako must have a *replica* function defined if programmers want to use the function assignment operator in this way.

Finally, the function assignment operator can be used with primitive types

```
int a := 3;
int b := 4;
a := b;
```

The above Tako code is translated to Java as

```
int a = 3;
int b = 4;
a = b;
```

In Tako, primitive types are meant to be treated conceptually as objects, and therefore can be considered to have an implicit replica function defined in them. For efficiency reasons the above Java translation is used when primitive types are used. In the current revision of Tako the distinction between primitive types and user defined types is not completely eliminated. Future versions of the compiler will fix this.

4.3.3 Parameter Passing

The implementation requirements for in-out parameter passing in Tako were as follows:

- All parameters by default are in-out unless specified otherwise by *eval* mode.
- The implementation should not change the original signature of the function
- The implementation should work seamlessly with other Java classes (integration issue)

- The variable *this* is also an implicit in-out parameter (except for static methods).

- The implementation should allow actual parameters of the method to be subtypes of the formal parameters, as in Java

- The formal parameters when passed back to the actual parameters should not cause any type violations (run time errors).

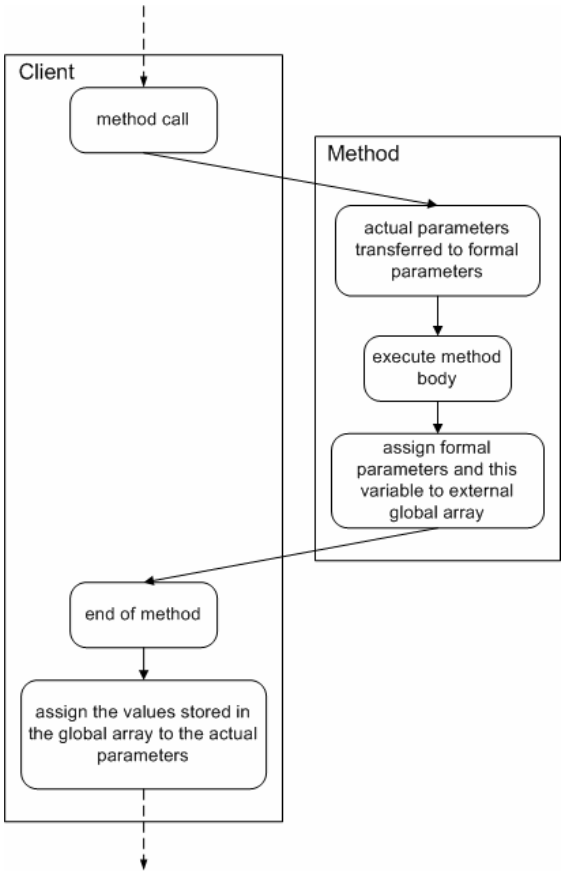

Figure 4.6 Parameter passing in Tako

The Tako compiler uses a scheme similar to the "boxing with refinement" method for parameter passing described in [Gou 98]. In this implementation, the caller and the called

method have access to an external array for communicating the parameter values. The method stores the formal parameters in this external array before returning back to the caller. The caller retrieves these stored parameters and assigns them to the actual parameters thus achieving in-out parameter passing. Figure 4.6 illustrates this. This technique allows for actual parameters to be subtypes of the formal parameters as in java. But problems can occur when the objects are assigned back to the actual parameters. This can be seen from the following example

```
Cat c;
Dog d;
c.exchange(d);
```

Where the *exchange* method is defined in a base class *Animal* as:

Tako version of *exchange*	**Java translation of *exchange***

`// exchange method defined in Animal`
`public void exchange (Animal x) {`
`  this :=: x;`
`}`
	`// exchange method defined in Animal`
	`public void exchange(Animal x) {`
	`  Animal $this = this;`
	`  Animal temp = $this;`
	`  $this = x;`
	`  x = temp;`
	`  TKGlobals.params[0] = $this;`
	`  TKGlobals.params[1] = x;`
	`}`

Figure 4.7 exchange method in Tako (left) and Java translation (right)

The above code tries to exchange *c* and *d* which are of type *Cat* and *Dog* respectively. A naïve implementation of this method will cause a runtime error because on returning from the method it will attempt to assign a runtime *Dog* object to a variable of type *Cat*, and it will attempt to assign a runtime *Cat* object to a variable of type *Dog*. To overcome this issue, when the formal parameters are assigned back to the actual parameters, we implement safe casting check to ensure that the assigned object is of the same type or subtype of the actual parameter. If it is not then default values are assigned to the actual parameters. Based on this scheme, the Java translation of the above code is shown in Figure 4.8.

In the translated Java code the objects in the parameter array are checked to ascertain that they can be assigned to the actual parameters. This is done using the *instanceof* operator in Java. The actual parameter is initialized if the object cannot be assigned to it. This avoids any type inconsistencies that may occur when the method is executed and the parameters are transferred back.

Tako code	Java translation

`Cat c;  // cat inherits from Animal`
`Dog d;  // dog inherits from Animal`
`c.exchange(d);`
	`Cat c = new Cat();`
	`Dog d = new Dog();`
	`c.exchange(d);`
	
	`if(TKGlobals.params[0] instanceof Cat) {`
	`  c = (Cat)TKGlobals.params[0];`
	`}`
	`else {`
	`  c = new Cat();`
	`}`
	
	`if(TKGlobals.params[1] instanceof Dog) {`
	`  d = (Dog)TKGlobals.params[1];`
	`}`
	`else {`
	`  d = new Dog();`
	`}`

Figure 4.8 Parameter passing in Tako (left) and Java translation (right)

Even though this strategy for parameter passing satisfies all the requirements enlisted, certain limitations exist for the usage of this implementation. This scheme is not meant to be used in conjunction with multiple threads. All threads will compete for access to this external parameter array causing inconsistencies in the parameter values. Access to this external array could be serialized thus avoiding any inconsistency, but this may lead to poorer overall performance due to the bottleneck caused by access to a single external array by all the threads. A solution to this problem could be to maintain individual parameter arrays for each thread.

The present version of the compiler does not have this implemented; later versions will include this fix.

Function Decomposition

Implementation of parameter passing for methods with no return value is straightforward as each statement will only contain a single method call. This allows for the parameter passing code to be added immediately after the statement, ensuring that all side-effects to the parameters are reflected immediately after the call. On the other had, parameter passing for side-effecting methods that also returns a value is not as trivial as with methods with no return value. Method calls could be nested, resulting in more than one method call per statement. In such cases, parameter passing code for each method call can be added only in the next statement. This does not guarantee that the changes to the parameters for the nested methods are reflected immediately after the call. To illustrate this point in more detail consider the following side-effecting function.

```
public Stack combineWith( Stack s2)
  ensures result = #this ○ #s2 and |this| = 0 and |s2| = 0;
```

The above function appends the stack *s2* to the current stack ***this*** and returns the combined stack as the result of the function. The side-effect of the function is that both the current stack (***this***) and *s2* are emptied. Consider the usage of the above function in the following Tako code.

```
// s1 = <1, 2, 3> and s2 = <4, 5, 6>
Stack s3 := s1.combineWith(s1.combineWith(s2));
```

Two stacks *s1* and *s2* have elements *<1, 2, 3>* and *<4, 5, 6>* respectively. A third stack *s3* is created from *s1* and *s2* using two *combineWith* method calls, one nested inside the other. A simplistic implementation of the array-based parameter passing scheme would not update parameters *s1* and *s2* when the inner function call is executed. This would result in the inner function call returning *<1, 2, 3, 4, 5, 6>* and *s1* and *s2* remaining unchanged. Therefore when the outer function call is executed, the old value of *s1* which is *<1, 2, 3>*, would be combined

with *<1, 2, 3, 4, 5, 6>*. This would result in the third stack being incorrectly created as *<1, 2, 3, 1, 2, 3, 4, 5, 6>*. Therefore, a more sophisticated implementation is needed to first update the parameter values before calling the outer function. The correct contents of stack *s3* should be *<1, 2, 3, 4, 5, 6>*.

To overcome this deficiency in the simplistic parameter passing implementation, the nested function calls are decomposed to individual statements. The manner in which the function calls are decomposed depends on the occurrence and depth of the nested calls. The innermost calls are invoked before the outer calls into separate statements. If there is more than one nested call on the same nesting level, then the function call which is encountered first is executed first. The return values of the calls are stored in a temporary function array to reconstruct the original method call. Thus, the last statement in the code is decomposed as:

```
TKGlobals.func[0] = s1.combineStack(s2);
/* parameter passing code for s1 and s2 */
s1.combineStack(TKGlobals.func[0])
/* parameter passing code for only s1 */
Stack s3 = (Stack)TKGlobals.result;
```

In this implementation, all the parameters are updated appropriately after each call, resulting in expected behavior of the nested function calls. This decomposition of nested function calls is done by a separate visitor in the compiler. This visitor is run before the parameter passing visitor to ensure that only one function call exists per statement.

Eval mode

In Tako, all parameters are in-out by default. A parameter can also be in eval mode. The eval keyword is used in Tako to indicate that a function is expected as an argument.

```
public int sum (eval int a, eval int b);
```

In the compiler *eval* mode is implemented using annotations. Annotations are a Java 1.5 feature that allows marking code in Java with tags. These tags are embedded into the compiled class files also. This allows the compiler to check for *eval* parameters even when the source files are

not present. The *eval* parameters are thus tagged with a *@Eval* annotation to allow the compiler to identify them.

During analysis, the parameters of all method calls in the code are checked to see if they are in eval mode. The parameters in eval mode expect a function as an argument. If a variable *a* is provided to a parameter which is in eval mode, it is translated as *a.replica()*. Eval mode is typically used for small types such as Booleans and int's whose copies can be made easily.

Repeated Arguments

Parameter passing can still cause aliasing in Tako due to repeated arguments as discussed in [Kul 05]. In Tako this is solved by initializing subsequent repeated arguments in the method call. However not all arguments can be identified as repeats at compile time. This occurs, for example, when using arrays as arguments. The statement *a[i].append(a[j])* will have repeated arguments only when $i = j$. This can be determined (and avoided) at runtime. To handle the different cases for repeated arguments during parameter passing the compiler uses the algorithm outlined below in Figure 4.9.

```
for each parameter passed in the function call
if the current parameter is potential repeat from previous parameters then
        if the parameter is a confirmed repeat then
            pass initial value instead of the actual parameter
        else
            Construct a conditional runtime check
            if the conditional check passes then
                Pass initialized value as it is a repeat
            else
                keep parameter unchanged
            endif
        endif
    else
        keep parameter unchanged
    endif
end loop
```

Figure 4.9 Algorithm for repeated arguments

The algorithm checks if each parameter is a potential repeat. A parameter is a potential repeat if it conflicts with previous parameters. A confirmed repeat occurs if the conflict does not

depend on some runtime condition. For example, in the function call *f(a, a.q)* the variable *a.q* is a potential repeat since *a.q* conflicts with the previous parameter *a*. It is also a confirmed repeat because *a.q* definetly conflicts with *a* and no runtime information is needed to conclude this. On the other hand, in the function call *f(a[i], a[j])* the variable *a[j]* is a potential repeat because it conflicts with a[i] if $i = j$. In this case we cannot confirm that it is a repeat because this depends on a runtime condition (whether $i = j$). In such cases a conditional runtime check is performed at runtime and appropiately an initialized value is passed if needed.

The above algorithm does not completely conform with the formal view of parameter passing with repeated arguments in [Kul 05], but later versions of the compiler will correct this discrepancy.

4.3.4 Pointer component

Tako provides a pointer component to enable construction of structures which must use aliasing for efficiency reasons. The compiler implements the pointer component and it's syntax using Java references. This is done for efficiency reasons and ease of implementation. Thus, the Pointer component syntax is only syntactic sugar for Java statements with objects. Figure 4.10 shows the translation from the pointer component syntax to Java performed by the compiler.

Tako syntax	Java translation

`class Node is Object ^(next);`	`class Node extends PointerType {`
	`  Node next = null;`
	`  Object contents = new Object();`
	`}`

`allocate p;`	`p = new Node();`
`p -> q`	`p = q;`
`p -> q^next;`	`p = q.next;`
`p^next -> q;`	`p.next = q;`

`p *:=: s;`	`Object temp = p.contents;`
	`p.contents = s;`
	`s = temp;`

Figure 4.10 Pointer component syntax

Chapter 5

Formal Reasoning

It is well known that aliasing complicates formal reasoning [Hoa 75]. Hoare has given several reasons why references should be avoided in [Hoa 75], including (1) Programmer need to keep track of two values for a reference variable, the reference value and the object value; and (2) Programmers need to keep track of how many reference variables point to a single object value so as to keep track of updates made through these variables.

The following quote from Hogg et al. [Hog 91] emphasizes this reasoning problem with aliasing.

> "To the formalist, it can be annoyingly difficult to prove the simple Hoare formula {*x* = *true*} *y* := ***false*** {*x* = *true*}. If *x* and *y* refer to the same boolean variable, i.e., *x* and *y* are *aliased*, then the formula will not be valid, and proving that aliasing cannot occur is not always straightforward."

Since aliasing is a runtime property, the fact that *x* and *y* are aliases cannot be proved statically. Therefore, Müller states that aliasing is the reason why most real-world software cannot be formally verified [Mül 00].

Despite the disadvantages of references and aliasing, most object-oriented languages routinely employ aliasing and rely on reference semantics to reason about code. In these languages, variables denote reference to objects. Tako, on the other hand, avoids the most common source of aliasing and uses value semantics (variables denote objects). The impact of this difference can be illustrated in the formal specification. For example, in JML, a popular specification language for Java, a mathematical map can be represented in 9 different ways:

1. JMLEqualsToEqualsMap,
2. JMLEqualsToObjectMap,
3. JMLEqualsToValueMap,
4. JMLObjectToEqualsMap,
5. JMLObjectToObjectMap,
6. JMLObjectToValueMap,
7. JMLValueToEqualsMap,
8. JMLValueToObjectMap
9. JMLValueToValueMap.

In comparison, a mathematical map in Tako only has one form which would be similar to the JMLValueToValueMap – because all types are values.

5.1 Basic Proof rules

Proof rules for some basic Tako statements are given in Figure 5.1. They are similar to those given in [Kro 88]. These rules, and the proof rules for procedure calls that follow are used in the formal proofs of correctness given in the appendix.

Procedure body verification:

```
assume pre; assertive-body; confirm post;
/* spec of current class is in context */
void p(T2 y)
  requires pre; ensures post;
{ assertive-body; }
```

Variable declaration:

```
assertive-code; confirm Q[x ← new T];
assertive-code; T x; confirm Q;
```

Swapping:

```
assertive-code; confirm Q[x ← y, y ← x];
assertive-code; x :=: y; confirm Q;
```

Initializing transfer:

```
assertive-code; confirm Q[x ← y, y ← new T];
assertive-code; x <- y; confirm Q;
```

Figure 5.1 Proof rules for select Tako statements

The rules are "specification-aware", meaning that they are intended for use with a language that includes syntactic slots for assertions (specifications) such as RESOLVE [Sit 94] and JML [Lea 99]. The rules work by "reducing" the code beneath the line to the code above the line. Eventually, the code is reduced to an assertion in mathematical logic, which is true if and only if the original code conforms to it's specification.

The first rule in Figure 5.1 is used for proving the correctness of procedure body declarations. The precondition is assumed and the postcondition of the procedure needs to be confirmed. The *assertive-body* is the body of the procedure along with any assertions, such as loop invariants. The code has a *context* associated with it. In this case the context will contain the specifications of the types of the parameters of the method. Note that in this case the method has two parameters - the explicit parameter *y* and the implicit (current) parameter *this*.

The variable declaration rule adheres to Tako's schema of automatic initialization. This rule says that if we have a variable declaration of x followed by an assertion Q that we need to

confirm, we can reduce that to confirming the assertion Q where x is replaced by ***new*** T (an initial value for x)

The remaining rules are the data assignment rules for swapping and initializing transfer. In the swapping rule, we eliminate the swap statement and replace all occurrences of x with y and all occurrences of y with x in the assertion to be confirmed. Similarly, in the initializing transfer rule, we replace all occurrences of x with y and all occurrences of y with an initial value of its type.

Before we go onto the next section we demonstrate a very simple correctness proof for the *exchange* function declared in the *Animal* class seen previously in Figure 4.7. The exchange method ensures that after the execution of the method, the current object (*this*) will be equal to the value of the old incoming value of x (denoted by #x), and the variable x will have the value equal to the old current object (denoted by #*this*).

```
public void exchange (Animal x)
  ensures this = #x and x = #this
{
  this :=: x;
}
```

By applying the procedure verification rule to the above code, we can reduce it to the following assertive program. The proof rules are applied in this manner till we reduce the assertive program to an assertion in predicate logic.

```
assume true;
this :=: x;
confirm this = #x and x = #this;
```

Since the requires clause for *exchange* is empty there is an implicit *assume true;*, and we have to confirm the *ensures* clause. By the proof rule of the swapping statement in Figure 5.1, we can remove the swap statement and replace *this* with x and similarly x with *this* in the confirmation clause.

```
assume true;
confirm x = #x and this = #this;
```

Once the assertive code is reduced to an assume statement followed by a confirm statement, we can replace it by an implication in which the assume clause is the antecedent and the confirm clause is the consequent. We can remove the # marks in this step.

```
true ⇒ (x = x and this = this)
```

The above implication is clearly true, thus we have proved correctness for the *exchange* function. The next section describes the rules for procedure calls and function calls.

5.2 Simple procedure call rule

The rule for a simple procedure call is shown below. It is similar to the rule for procedure calls given in [Gri 80]. The rule assumes that the argument to the call are not repeated. It also assumes that *p* was declared in class *T1* as ***void*** *p(T2 x)*.

```
assertive-code; confirm (pre[this ← a, x ← b]
  and ∀?a: T1, ∀?b: T2,
  post[#this ← a, this ← ?a, #x ← b, x ← ?b])
  implies Q'[a ← ?a, b ← ?b];
assertive-code; a.p(b); confirm Q;
```

Broadly, the rule says that if you have a procedure call followed by an assertion *Q* that must be confirmed, you can replace it by a conditional assertion – if the precondition holds and the postcondition holds for all possible new values of the parameters, then the original assertion *Q* holds where *a* and *b* are replaced by the new parameter values.

Before we continue with a more sophisticated rule for procedure calls, we demonstrate how this rule can be used in a formal proof of correctness for the following code. The variables *a* and *b* are of type *Animal* and hold values *A* and *B* respectively.

```
assume a = A and b = B;
a.exchange(b);
confirm a = B and b = A;
```

By applying the proof rule for simple procedure call described previously for the *exchange* method the above reduces to the following.

```
assume a = A and b = B;
confirm (∀?a: Animal, ∀?b: Animal,
  ?a = b and ?b = a
  implies ?a = B and ?b = A);
```

This can be converted to the following implication

```
a = A and b = B ⇒ ∀?a: Animal, ∀?b: Animal,
                  ?a = b and ?b = a
                  implies ?a = B and ?b = A
```

Since $a = A$ and $b = B$, replacing these values in the consequent we get, $?a = B$ and $?b = A$ which is the same as the implies clause in the consequent. Thus we have proved the correctness of the above assertive code with a procedure call.

5.3 Sophisticated procedure call rule

The proof rules we have looked at thus far are variations of rules found elsewhere [Ogd 00, Kro 88]. Furthermore, a procedure call rule that considers repeated argument can be found in [Kul 05]. None of these rules however, take into consideration Java-style polymorphism and subtyping. We introduce the following generalized rule that takes into consideration both repeated arguments and subtyping. The proof rule assumes the following

1. p is declared in class T as ***void*** $p(T_x\ x)$
2. a is of type T_a and b is of type T_b, and
3. T_a <: T and T_b <: T_x. The symbol <: represents the subtype relationship, as described in [Lis 94].

```
assertive-code; T %a; Tx %b;
%a <- a; %b <- b;
%a.p(%b)';
b <-() %b; a <-() %a;
```

confirm Q;
assertive-code; a.p(b); **confirm** Q;

This rule suggests that the procedure call can be considered equivalent to the following series of sub statements. Initially two temporary local variables *%a* and *%b* of type *T* and T_x are created and initialized. The actual parameters *a* and *b* are transferred into them using the initializing transfer operation. This results in *%a* and *%b* being distinct as the arguments which are repeated are initialized when transferred in. Subsequent transfers result in an initial value being assigned. For example, in the procedure call *a.p(a)*, when *a* is transferred into *%a*, *a* will get an initial value. Therefore, when *a* is again transferred into *%b*, it gets assigned an initial value, thus avoiding any repeats. The initializing transfer also supports arguments that have a subtype relationship to the formal parameters. The procedure call is then made with *%a* and *%b* as parameters. This call has no repeated arguments and the parameters are of the same type as the formal parameters. Note that these are the same assumptions of the simple procedure call rule. The simple procedure call is denoted as *%a.p(%b)'*, indicating that the sophisticated procedure call (that allows subtyping and repeated arguments) can be reduced to and expression using the simple procedure call (that does not allow subtyping and repeated arguments).

After the procedure call, the values in the parameters *%a* and *%b* are transferred back to *a* and *b*. This is done using the safe casting operator denoted by `<-()`. This operator behaves as an initializing transfer operator if the runtime type of the variable on the right hand side of the operator is compatible with the static type of the variable on the left hand side. If this is not true, then both the variables involved in the operation get initialized. This takes into consideration the situations when the objects being passed out are incompatible with the types of the actual parameters. The proof rule for the safe casting operator is given below.

assertive-code; **confirm** (y.**class** <: T_x ⇒ Q[x ← y, y ← **new** T_y]) **and**
(**not** y.**class** <: T_x ⇒ Q[x ← **new** T_x, y ← **new** T_y])
assertive-code; x <-() y; **confirm** Q

The rule has two implications denoted by the two conjuncts. If the runtime type of *y* denoted by *y.class* is a subtype of T_x (the type of the formal parameter *x*), then the assertion Q needs to be confirmed with *x* substituted by *y* and *y* substituted by *new* T_x (an initial value of type T_x). If this condition is not true, then Q needs to be confirmed with both *x* and *y* replaced with initial values. As indicated <: denotes the subtyping relation. A class *A* is a subtype of class *B* if class *B* respects the invariants in the supertype *A*, the subtype methods preserve the supertype method's behavior, and the subtype constraints ensure supertype constraints.

The application of the above procedure call rule is illustrated in the Appendix. A syntactic variation of the procedure call rule is the function call rule which handles side effecting function assignment statements such as *x := s.pop()*, where the *pop* function call modifies the current stack object *this*. The function call rule is also applicable for functions which restore the value of the formal parameters including *this* (i.e. #*this* = *this*). The rule is given below.

```
assertive-code; T1 %a; T2 %b;
%a <- a;
%b := %a.f()';
b <-() %b; a <-() %a;
confirm Q;
assertive-code; b := a.f(); confirm Q;
```

Similar to the procedure call rule, the function call *%b := %a.f()'* indicates that the following simple function call proof rule is applied to reduce the statement.

```
assertive-code; confirm pre[x ← a, y ← b]
  and ∀?a: T1, ∀?b: T2,
  post[#this ← a, this ← ?a, result ← ?b ]
  implies Q'[a ← ?a, b ← ?b];
assertive-code; b := a.f(); confirm Q;
```

Chapter 6

Case Study

To experience first hand the paradigm shifts involved in programming a non-trivial application in Tako, we undertook the development of a text-based adventure game. The game was initially developed in Java, but with the intent that it would eventually be ported to Tako. To give an idea of the complexity of the application, the Figure 6.1 gives and overview of its design.

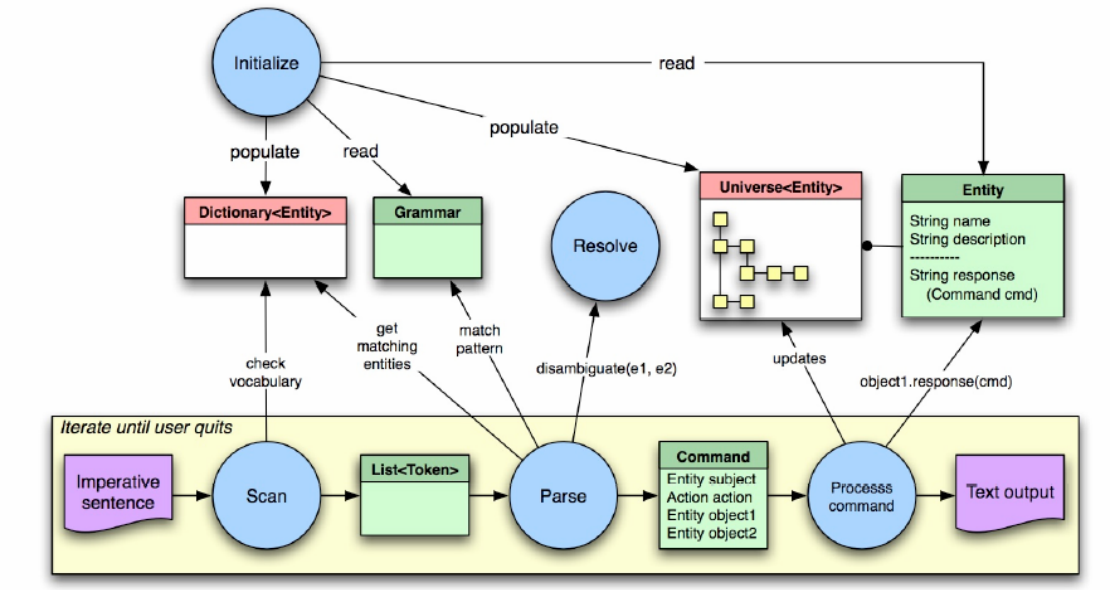

Figure 6.1 Adventure game design

The game accepts text inputs from the user which are usually simple imperative sentences, such as "take the chess piece" or "put the red queen on the chess board". The game has a parser component which parses this input based on a supplied grammar and dictionary of game objects. The game also has a "resolver" component that tries to determine what game

object is intended when the player enters ambiguous text. Apart from processing the input, the other important aspect of the game is the universe. The universe is a tree-like data structure that stores all the game objects. When the user inputs a command, the application updates the universe accordingly.

Metrics for the Java program and the Tako program are given in Figure 6.2.

Java metrics	Tako metrics
49 classes	52 classes
4164 lines of code	4012 lines of code
6 packages	6 packages

Figure 6.2 Metrics for adventure program

The program contained about 50 classes and consisted of over 4000 lines of code. It required approximately 85 man hours to code the game in Tako based on the Java version of the game. Figure 6.3 gives rough breakdown of how the hours where spent.

Description	Hours
Time spent during translation	
Conversion of enum types in Java to static integer variables in Tako	2
Converting side effecting methods with return values	10
Converting non-side effecting methods with return values	5
Translating code containing aliasing to the swapping methodology	15
Simple translations (copy and paste)	5
Time spent after translation	
Debugging errors which occurred due to erroneous translation	30
Debugging errors already present in the original Java version	18

Figure 6.3 Breakup of hours

We can divide the hours into time spent during the translation process and the time spent after the translation process as shown in Figure 6.3. During the translation process we spent the maximum amount of time translating statements that contained the use reference copy operation into statements that replaced it with the swapping operation. This process did not involve simply substituting the reference copy operator with the swap operator; we also had to remember to swap the objects back, if needed. We also spent a fair amount of time converting methods with return values to their equivalent in Tako. If the methods had side-effects, then they where changed to procedures (methods without return values) in Tako. The return value in procedures is passed through the parameters. If the methods did not have side-effects, then the methods where changed to appropriate functions (methods with return values) in Tako. In Tako functions, the distinguished "result" variable is used to store the return value. Some time was also spent for converting enum types to static integer variables. Enum types are supported in Java 1.5 but not in Tako. The remaining time was spent in copy and paste kind of operations (due to the similarities between Java and Tako syntax).

A majority of the time was also spent after the translation process in debugging the code. The debugging process could be split up into two parts. Time spent in debugging errors that occurred due to erroneous translation and the time spent in errors that where present in the original version of the Java code. Nearly 30 hours were spent in debugging the translation errors. It was expected that maximum amount of time would be spent in this process as this was our first full fledged application in Tako. The other 18 hours spent in debugging could have been avoided if the original Java version was tested thoroughly.

6.1 Paradigm shift

While designing and implementing the code we observed a few things about the paradigm shift involved. These points are discussed in this section.

A typical Java version of this game might use the composite design pattern to implement the game object and universe structure. While the composite pattern can be implemented in Tako,

the recursive type structures involved raise the possibility that null references will be assigned to variables, which, in general, is something we prefer to avoid in Tako. Therefore, we developed a tree like data structure to represent the universe. Since the composite pattern is effectively a tree data structure, a well designed tree component could help programmers to design software that does not depend on the composite pattern. Note that in this case study even the original Java code contained the tree data structure.

Another thing we noticed was that in this particular program the distinction between object identity and unique name identity did not play a major role. We used hash maps to store the game objects and each game object had a unique key associated with it. In both the Java and Tako version, it was these unique keys rather than the language dependent object identity that was used for uniquely identifying the objects.

Since all the game objects are stored in a tree like data structure, accessing these objects in Tako meant swapping them out of the structure, examining them, and then swapping them back in. In the Java version, programmers modify a game object through a handle or reference to the object. The swapping in Tako initially lead to errors while inserting them back in the tree as the conceptual cursor position in the tree was unexpectedly modified. The error was easily fixed, but it represents one example of an error that would not occur in Java as the object is never removed in the first place.

In the Java version of the game, the container classes like stacks, hash maps, queues, and lists were already provided by the *java.util* package. But in the Tako version, these util classes where implemented from scratch using the pointer component. The pointer component was only used in these low level classes; while the higher design level classes did not require the use of the pointer component. This supported our conjecture that the programmer, at higher level of design, can make do without using references.

Not only did we successfully implement the game, we were also able to use the Tako game classes along with Java graphical interface classes to give it a GUI based on the Java Swing components.

Overall, we found that the paradigm shifts involved were not very difficult to adjust to though they required some alertness in terms of the swapping paradigm. This experience is consistent with that of Hollingsworth et al., who in [Hol 00] discusses the implementation of a commercial application developed in C++ using the Resolve discipline, which has many similarities with Tako. It consisted of about 250 components and 100,000 lines of C++ code. In their report, they concluded that swapping worked well with most object-oriented techniques.

Chapter 7

Conclusion

In this thesis we have described the design and implementation of the Tako language and compiler. We looked at how some of the interesting features, such as in-out parameter passing, automatic initialization, and repeated arguments, were implemented. We also developed a non-trivial application in Tako, which exposed the paradigm shifts involved in moving from Java to Tako. Finally, this thesis introduced new proof rules for procedure calls which took into consideration both repeated arguments and Java-style subtyping.

The proof rule is based on value semantics, which indicates that alias-avoidance techniques can be incorporated into an object-oriented language in a way that preserves Java-style inheritance and polymorphism. The similarities between the Java and Tako version of the text adventure application indicates that Java programmers can adjust to the paradigm shifts involved in using alias avoidance techniques based on the swapping paradigm.

Ultimately, we would like to have a complete value semantics defined for Tako. Furthermore, a thorough exploration of the different paradigms between Java and Tako would require input from other Java programmers. Other future work would involve the development of a specification and verification system for Tako programs. This would include the development of a Tako specification language which would be similar to JML or RESOLVE and the integration of the compiler with a theorem prover to prove the correctness of programs.

Bibliography

[Aba 97] Abadi, M. and Leino, K.R.M. A logic of object-oriented programs. Dauchet, M. ed. TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference, pages 682-696. Springer-Verlag, New York, 1997.

[Ald 02] Aldrich, J., Kostadinov, V. and Chambers, C., Alias annotations for program understanding. *In Proceedings 17th ACM Conference (OOPSLA '02)*, pages 311-330. ACM Press, 2002.

[Alm 97] Almeida, P.S., Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP '97*, pages 32-59. Springer-Verlag, New York, 1997.

[Bak 95] Baker, H.G. 'Use-once' variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, *30* (1). pages 45-52. 1995.

[Bok 99] Bokowski, B. and Vitek, J., Confined types. In *Proceedings 14th ACM SIGPLAN Conference OOPSLA '99*, pages 82-96. ACM Press, 1999.

[Bor 93] Borgida, A., Mylopoulos, J. and Reiter, R., ... And nothing else changes?: The frame problem in procedure specifications. In *Proceedings of the 15th International Conference on Software Engineering*, pages 303-314. IEEE Computer Society Press, 1993.

[Cha 05] Chalin, P. and Rioux, F., Non-null references by default in the Java Modeling Language. In *Proceedings Specification and Verification of Component-Based Systems*, pages 70-76. 2005.

[Che 05] Cheon, Y., Leavens, G.T., Sitaraman, M. and Edwards, S. Model variables: Cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, *35* (6), pages 583-589. 2005.

[Cla 98] Clarke, D.G., Potter, J.M. and Noble, J., Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, pages 48-64, ACM Press, 1998.

[Cla 03] Clarke, D.G. and Wrigstad, T., External uniqueness. In *Proceedings FOOL 10*, 2003.

[Fla 02] Flanagan, C., Rustan, K., Leino, M., Lillibridge, M., Nelson, G., Saxe, J.B. and Stata, R., Extended static checking for Java. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234-245. ACM Press, 2002.

[Gam 94] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns*. Addison-Wesley, 1994.

[Gos 05] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification*. Addison-Wesley, 2005.

[Gou 98] Gough, J., *Parameter Passing for the Java Virtual Machine*. In Proceedings of the Australasian Computer Science Conference, 1998.

[Gri 80] Gries, D. and Levin, G. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems*, *2* (4), pages 564-579. 1980.

[Har 91] Harms, D.E. and Weide, B.W. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, *17* (5). pages 424-435. 1991.

[Hoa 71] Hoare, C.A.R., Procedures and parameters: An axiomatic approach. In *Proceedings Symposium on Semantics of Algorithmic Languages*, pages 102-116. Springer-Verlag, 1971.

[Hoa 75] Hoare, C.A.R., Recursive data structures. *International Journal of Computer and Information Sciences 4*, 2, pages 105-132, 1975

[Hog 91] Hogg, J., Islands: Aliasing protection in object-oriented languages. In *Proceedings OOPSLA '91*, pages 271-285. ACM, 1991.

[Hog 92] Hogg, J., Lea, D., Wills, A., deChampeaux, D. and Holt, R. The geneva convention on the treatment of object aliasing. *OOPS Messenger*, *3* (2). pages 11-16. 1992.

[Hol 00] Hollingsworth, J.E., Blankenship, L. and Weide, B.W., Experience report: Using Resolve/C++ for commercial software. In *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, pages 11-19. ACM Press, 2000.

[Kho 86] Khoshaflan, S.N. and Copeland, G.P., Object identity. In *OOPSLA '86 Conference Proceedings*, pages 406-416. ACM Press, 1986.

[Kie 76] Kieburtz, R.B., Programming without pointer variables. In *Proceedings of the SIGPLAN '76 Conference on Data: Abstraction, Definition, and Structure*, ACM Press, 1976.

[Kro 88] Krone, J. *The Role of Verification in Software Reusability*, Doctoral Thesis, The Ohio State University, 1988.

[Kul 05] Kulczycki, G., Sitaraman, M., Ogden, W.F. and Weide, B.W. *Clean Semantics for Calls with Repeated Arguments*, Technical Report RSRG-05-01, Clemson University, 2005.

[Kul 05a] Kulczycki, G., Sitaraman, M., Weide, B. and Rountev, N., A specification-based approach to reasoning about pointers. In *Proceedings Specification and Verification of Component-Based Systems*, pages 55-62. 2005.

[Lea 99] Leavens, G.T., Baker, A.A. and Ruby, C. JML: A notation for detailed design. Simmonds, I. ed. *Behavioral Specifications of Businesses and Systems*, Kluwer, 1999.

[Lea 02] Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C. and Cok, D.R., How the design of JML accommodates both runtime assertion checking and formal verification. In *Formal Methods for Components and Objects: First International Symposium*, 2002.

[Lei 98] Leino, K.R.M., Data groups: Specifying the modification of extended state. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 144-153, ACM Press, 1998.

[Lei 00] Leino, K. R. M., Nelson, G., Saxe, J. B., *ESC/Java User's Manual. Technical Note 2000-002*. Compaq Systems Research Center, 2000.

[Mül 00] Müller, P. and Poetzsch-Heffter, A. Modular specification and verification techniques for object-oriented software components. Sitaraman, M. and Leavens, G. eds. *Foundations of Component-Based Systems*, Cambridge University Press, Cambridge, United Kingdom, 2000.

[Min 96] Minsky, N.H., Towards alias-free pointers. In *Proceedings ECOOP '96*, pages 189-209. 1996.

[Nob 98] Noble, J., Vitek, J. and Potter, J. Flexible alias protection. *Lecture Notes in Computer Science*, *1445*. pages 158-185. 1998.

[Ogd 00] Ogden, W.F. *The Proper Conceptualization of Data Structures*. The Ohio State University, Columbus, OH, 2000.

[Ohe 01] O'Hearn, P., Reynolds, J. and Yang, H. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, *2142*, 2001.

[Pop 77] Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G. and London, R.L. Notes on the design of Euclid. *ACM SIGPLAN Notices*, *12* (3), poages 11-18. 1977.

[Sit 00] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W., Pike, S. and Hollingsworth, J.E., Reasoning about software-component behavior. In *Procs. Sixth Int. Conf. on Software Reuse*, pages 266-283. Springer-Verlag, 2000.

[Sit 94] Sitaraman, M. and Weide, B.W. Component-based software using Resolve. *ACM Software Engineering Notes*, *19* (4), pages 21-67. 1994.

[Van 01] van den Berg, J., Jacobs, B., *The LOOP Compiler for Java and JML*. n: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer, 2001.

[Wei 01] Weide, B.W. and Heym, W.D., Specification and verification with references. In *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*. 2001.

[Wei 02] Weide, B.W., Pike, S.M. and Heym, W.D., Why swapping? In *Proceedings RESOLVE 2002 Workshop*, 2002.

Appendix A

This appendix contains the following two proofs:

1. A proof of correctness of the *reverse* method in the *Stack* class.

2. A proof of the client code with the procedure call *exchange* described in Figure xx with repeated arguments and subtyping.

Proof #1: A proof of correctness of the *reverse* method in the *Stack* class. The implementation of the reverse method is based on the unbounded Stack illustrated in Figure 4.4.

```
public void reverse()
  ensures this = REV(#this);
{
  Object x;
  Stack temp;
  while (this.length() != 0)
    invariant REV(temp) o this = #this;
  {
    this.pop(x);
    temp.push(x);
  }
  this :=: temp;
}
```

By applying the procedure verification rule (as in Figure 5.1) to the above code, we can reduce it to the following assertive program. We keep applying proof rules in this manner until we reduce the assertive program to an assertion in predicate logic. If the assertion is true, then correctness is proved.

```
assume true;
Object x;
Stack temp;
while (this.length() != 0)
  invariant REV(temp) o this = #this;
{
  this.pop(x);
  temp.push(x);
}
```

```
this :=: temp;
confirm this = REV(#this);
```

By the proof rule for swapping statement in Figure 5.1 we can remove the swap statement and replace *this* with *temp* in the confirmation clause.

```
assume true;
Object x;
Stack temp;
while (this.length() != 0)
    invariant REV(temp) o this = #this;
{
    this.pop(x);
    temp.push(x);
}
confirm temp = REV(#this);
```

At this point, we need to use a proof rule for the while loop, which we present below.

```
assertive-code;
confirm I;
assume I and B; body; confirm I;
(I and ¬B)⇒Q;
assertive-code;
while (B) invariant I; { body } confirm Q;
```

The rule reduces the assertive code to two assume-confirm blocks and one logical implication. These represent, in order, an initialization property, a maintenance property, and a termination property. This is shown below.

1. Initialization

```
assume true;
Object x;
Stack temp;
confirm REV(temp) o this = #this;
```

2. Maintenance

```
assume REV(temp) o this = #this and this.length() != 0;
this.pop(x);
temp.push(x);
confirm REV(temp) o this = #this;
```

3. Termination

```
REV(temp) ○ this = #this and
this.length() = 0 ⇒ (temp = REV(#this));
```

In the above three blocks we can replace *this.length*() by |*this*| because the ensures clause for *length* guarantees that the result of the expression *this.length*() is |*this*|.

1. Initialization

```
assume true;
Object x;
Stack temp;
confirm REV(temp) ○ this = #this;
```

2. Maintenance

```
assume REV(temp) ○ this = #this and |this| != 0;
this.pop(x);
temp.push(x);
confirm REV(temp) ○ this = #this;
```

3. Termination

```
REV(temp) ○ this = #this and
|this| = 0 ⇒ (temp = REV(#this));
```

The above three blocks must be proved true to discharge the proof. We will first start with the Maintenance block since it is typically the most difficult.

```
assume REV(temp) ○ this = #this and |this| != 0;
this.pop(x);
temp.push(x);
confirm REV(temp) ○ this = #this;
```

By the proof rule for simple procedure call (which does not take into consideration repeated arguments and subtyping) on *push* this reduces to the following.

```
assume REV(temp) ○ this = #this and |this| != 0;
this.pop(x);
confirm ∀?temp_s : Stack, ∀?x_s : Object;
        ?temp_s = x ○ temp
        implies REV(?temp_s) ○ this = #this;
```

Applying the same proof rule for procedure on *pop* we get

```
assume REV(temp) ○ this = #this and |this| != 0;
confirm |this| > 0 and
        ∀?this_s : Stack, ∀?x_t : Object
        this = ?x_t ○ ?this_s
        implies (∀?temp_s : Stack, ∀?x_s : Object
                 ?temp_s = ?x_t ○ temp impies
                 REV(?temp_s) ○ ?this_s = #this );
```

The second conjunct in the *confirm* clause has the form (A ⇒ (B ⇒ C)), which is logically equivalent to ((A and B) ⇒ C). We make this change, and also take the opportunity to group the quantifiers at the beginning of the second conjunct.

```
assume REV(temp) ○ this = #this and |this| != 0;
confirm |this| > 0 and
∀?this_s: Stack, ∀?x_t: Object, ∀?temp_s: Stack, ∀?x_s: Object
(this = ?x_t ○ ?this_s) and (?temp_s = ?x_t ○ temp)
implies REV(?temp_s) ○ ?this_s = #this;
```

Since we have a *assume* followed by a *confirm* we can replace it by an implication in which the *assume* is the antecedent and the *confirm* is the consequent.

```
(REV(temp) ○ this = #this and |this| != 0) ⇒
    |this| > 0 and
∀?this_s: Stack, ∀?x_t: Object,
∀?temp_s: Stack, ∀?x_s: Object,
(this = ?x_t ○ ?this_s) and (?temp_s = ?x_t ○ temp)
implies REV(?temp_s) ○ ?this_s = #this;
```

The quantifiers can be placed outside the implication to get

```
∀?this_s: Stack, ∀?x_t: Object,
∀?temp_s: Stack, ∀?x_s: Object,
(REV(temp) ○ this = #this and |this| != 0) ⇒ |this| > 0 and
(this = ?x_t ○ ?this_s) and (?temp_s = ?x_t ○ temp)
implies REV(?temp_s) ○ ?this_s = #this;
```

This is of the form

```
∀?this_s: Stack, ∀?x_t: Object,
∀?temp_s: Stack, ∀?x_s: Object,
outer_antecedent ⇒ (|this| > 0 and (inner_antecedent ⇒ consequent))
```

This is logically equivalent to

```
∀?this_s: Stack, ∀?x_t: Object, ∀?temp_s: Stack, ∀?x_s: Object
( outer_antecedent ⇒ |this| > 0 ) and
( ( outer_antecedent and inner_antecedent ) ⇒ consequent )
```

Thus we have to prove the following two assertions

```
1.  REV(temp) o this = #this and |this| != 0 ⇒|this| > 0

2.  (REV(temp) o this = #this) and |this| != 0 and (this = ?x_t o  ?this_s)
    and (?temp_s = ?x_t o temp) ⇒ REV(?temp_s) o ?this_s = #this
```

The first assertion is true because from the *outer_antecedent* we know that $|\boldsymbol{this}| \neq 0$ and $|\boldsymbol{this}|$ cannot be negative, so $|\boldsymbol{this}| > 0$.

In the second assertion we can simplify the string equality $REV(temp)$ o $\boldsymbol{this} = \#\boldsymbol{this}$ as follows:

```
REV(temp) o this = #this
```

Replace ***this*** with $?x_t$ o $?this_s$ from the *inner_antecedent*

```
REV(temp) o ?x_t o ?this_s = #this
```

From the *inner_antecedent* we have the string equality $?temps = ?x_t$ o $temp$. By string manipulation we get $REV(?temps) = REV(temp)$ o $?x_t$. Replacing the value of $REV(temp)$ o $?x_t$ in the above equality we get the following.

```
REV(?temp_s) o ?this_s = #this
```

This is the same as the consequent of the second assertion. Therefore the second assertion is also true and thus we have proved the maintenance block. It remains to show that the initialization and termination blocks reduce to true assertions. Recall that the initialization block is given as

```
assume true;
Object x;
Stack temp;
confirm REV(temp) o this = #this;
```

By the proof rule for variable declaration in Figure 5.1, we can replace *temp* with ***new*** *Stack* in the confirm clause.

```
assume true;
Object x;
confirm REV(new Stack) o this = #this;
```

Similarly by proof rule for variable declaration, we can replace all occurrence of *x* in the confirm clause with ***new*** *Object*. Since there is no *x* in the confirm clause, it remains the same.

```
assume true;
confirm REV(new Stack) o this = #this;
```

The above is converted to the following implication

```
true ⇒ (REV(new Stack) o this = #this)
```

By the ensures clause on the stack constructor, the value of ***new*** *Stack* is *EMPTY_STRING*. The reverse of an empty string is an empty string, so the consequent of the assertion becomes ***this*** = #***this***, which is true. Thus, we have proved the initialization property.

Finally, we prove the termination block.

```
REV(temp) o this = #this and |this| = 0 ⇒ (temp = REV(#this));
```

If |***this***| = 0, then ***this*** is an empty string, and therefore *REV*(*temp*) o ***this*** = #***this*** is equivalent to *REV*(*temp*) = #***this***, which is equivalent to *temp* = *REV*(#***this***). Therefore the assertion is true and we have proved termination.

Since the initialization, maintenance, and termination blocks were all reduced to true assertions, the reverse method has been proved correct with respect to its specification.

Proof #2: A proof of the client code with the procedure call *exchange* with repeated arguments and subtyping.

We want to prove the following assertive program:

```
assume true;
Cat c;
Dog d;
c.exchange(d);
confirm c = new Cat and d = new Dog;
```

By the proof rule of sophisticated procedure calls (for repeated arguments and subtyping) described in Section 5.3, on *exchange* this reduces to the following.

```
assume true;
Cat c;
Dog d;
Animal %c;
Animal %d;
%c <- c;
%d <- d;
%c.exchange(d)'
d <-() %d;
c <-() %c;
confirm c = new Cat and d = new Dog
```

By the proof rule for safe casting described in Section 5.3, we get the following.

```
assume true;
Cat c;
Dog d;
Animal %c;
Animal %d;
%c <- c;
%d <- d;
%c.exchange(d)'
d <-() %d;
confirm (%c.class <: Cat ⇒ %c = new Cat and d = new Dog) and
        (not %c.class <: Cat ⇒ new Cat = new Cat and d = new Dog);
```

Applying the same safe casting rule again

```
assume true;
Cat c;
Dog d;
Animal %c;
Animal %d;
%c <- c;
%d <- d;
%c.exchange(d)'
confirm (%d.class <: Dog ⇒
              ((%c.class <: Cat ⇒ %c = new Cat and %d = new Dog) and
               (not %c.class <: Cat ⇒ new Cat = new Cat and %d = new Dog))
         not %d.class <: Dog ⇒
              ((%c.class <: Cat ⇒ %c = new Cat and new Dog = new Dog) and
```

```
                (not %c.class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
        );
```

Since the procedure call *%c.exchange(d)'* does not have any repeated arguments and any subtypes, the proof rule for simple procedure call can be use to reduce this statement as shown below.

```
assume true;
Cat c;
Dog d;
Animal %c;
Animal %d;
%c <- c;
%d <- d;
confirm ∀?%c : Animal, ∀?%d : Animal
        ?%c = %d and ?%d = %c
        implies
         (?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and ?%d = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and ?%d = new Dog))
          not ?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and new Dog = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
         );
```

By the proof rule for initializing transfer in Figure xx, we can remove the transfer statement and replace *%d* with *d*.

```
assume true;
Cat c;
Dog d;
Animal %c;
Animal %d;
%c <- c;
confirm ∀?%c : Animal, ∀?%d : Animal
        ?%c = d and ?%d = %c
        implies
         (?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and ?%d = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and ?%d = new Dog))
          not ?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and new Dog = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
         );
```

Applying the initializing rule again we get.

```
assume true;
Cat c;
```

```
Dog d;
Animal %c;
Animal %d;
confirm ∀?%c : Animal, ∀?%d : Animal
        ?%c = d and ?%d = c
        implies
         (?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and ?%d = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and ?%d = new Dog))
          not ?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and new Dog = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
         );
```

By the proof rule for variable declaration in Figure 5.1, we can remove the statements *Animal %c;* and *Animal %d;*.

```
assume true;
Cat c;
Dog d;
confirm ∀?%c : Animal, ∀?%d : Animal
        ?%c = d and ?%d = c
        implies
         (?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and ?%d = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and ?%d = new Dog))
          not ?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and new Dog = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
         );
```

Similarly by the proof rule for variable declaration in Figure 5.1, we remove the statements *Cat c;* and *Dog d;* and replace all occurrences of *c* and *d* with *new Cat* and *new Dog* respectively.

```
assume true;
confirm ∀?%c : Animal, ∀?%d : Animal
        ?%c = new Dog and ?%d = new Cat
        implies
         (?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and ?%d = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and ?%d = new Dog))
          not ?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and new Dog = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
         );
```

The above can be converted to the following implication.

```
true ⇒ ∀?%c : Animal, ∀?%d : Animal
        ?%c = new Dog and ?%d = new Cat
```

```
        implies
         (?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and ?%d = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and ?%d = new Dog))
          not ?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and new Dog = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
         )
```

The implication above is of the form (*true* ⇒ (B ⇒ C)) which is logically equivalent to B ⇒ C. Making this change we get.

```
∀?%c : Animal, ∀?%d : Animal
?%c = new Dog and ?%d = new Cat ⇒
         (?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and ?%d = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and ?%d = new Dog))
          not ?%d.class <: Dog ⇒
              ((?%c.class <: Cat ⇒ ?%c = new Cat and new Dog = new Dog) and
              (not ?%c.class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
         )
```

From the implication we know that *?%c* = *new Dog* and *?%d* = *new Cat*, replacing these values in the consequent we get the consequent of the implication as follows.

```
(?(new Cat).class <: Dog ⇒
    ((?(new Dog).class <: Cat ⇒ new Dog = new Cat and new Cat = new Dog) and
    (not (new Dog).class <: Cat ⇒ new Cat = new Cat and new Cat = new Dog))
not (new Cat).class <: Dog ⇒
    (((new Dog).class <: Cat ⇒ new Dog = new Cat and new Dog = new Dog) and
    (not (new Dog).class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog))
)
```

It is clear that ***(new** Cat).**class*** is not a subtype of type *Dog*, and ***(new** Dog).**class*** is not a subtype of type *Cat*. Using this information we can reduce the above implication to the following.

```
not (new Cat).class <: Dog ⇒
       (not (new Dog).class <: Cat ⇒ new Cat = new Cat and new Dog = new Dog)
```

Which is true since the two antecedents as discussed before are obviously true, and it is trivial to see that ***new** Cat* = ***new** Cat* and ***new** Dog* = ***new** Dog* is true, hence we have proved the correctness of the code.

Appendix B

This appendix shows the Java translation given by the Tako compiler for two example Tako classes, *LinkedList* and *HashSet*. The *LinkedList* makes use of the pointer component as it is a linked data structure, while the *HashSet* class has no pointer component use, but makes use of the *LinkedList*.

Linked List (Tako code)

```
public class LinkedList  {

  class Node is TKObject ^(next);
  private Node head, pre, last;
  private int left_length, right_length;

  public LinkedList() {
   allocate head;
   pre -> head;
   last -> head;
  }

  public void insert(TKObject x) {
   Node post, new_pos;
   post -> pre^next;
   allocate new_pos;
   new_pos *:=: x;
   pre^next -> new_pos;
   new_pos^next -> post;
   if (right_length = 0) {
    last -> last^next;
   }
   right_length++;
  }

  public void remove(TKObject x) {
   //assert !atEnd();
   Node old_pos, new_post;
   new_post -> pre^next;
   old_pos -> new_post;
   new_post -> new_post^next;
   pre^next -> new_post;
   old_pos *:=: x;
   if (right_length = 1) {
    last -> pre;
   }
```

```
    right_length--;
   }

   public void advance() {
    //assert !atEnd();
    pre -> pre^next;
    left_length++;
    right_length--;
   }

   public void reset() {
    pre -> head;
    right_length := left_length + right_length;
    left_length := 0;
   }

   public void get(TKObject out) {
    if(pre^next != null ) {
     pre^next *:=: out;
    }
    else {
     System.out.println("pre^next is null");
    }
   }

   public int length() {
    result := left_length + right_length;
   }

   public boolean isEmpty() {
    result := (left_length + right_length = 0);
   }
}
```

Linked List (Java translation)

```
public class LinkedList extends tako.lang.TKObject {
  public class Node extends tako.lang.TKPointerType {

    private Node head = (Node)(null);
    private Node pre = (Node)(null);
    private Node last = (Node)(null);
    private int left_length;
    private int right_length;

    public Node replica() {
      tako.lang.TKObject $this = (tako.lang.TKObject)(this);
      java.lang.Object result = null;
      tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
      result = (java.lang.Object)(this);
      tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
      tako.lang.TKGlobals.result = result;
      return ((Node)tako.lang.TKGlobals.result);
    }
```

```
  public tako.lang.TKObject contents =
           ((tako.lang.TKObject)tako.lang.TKObject.create("tako.lang.TKObject"));
  public Node next = ((Node)tako.lang.TKObject.create("$NULL$"));
}

public LinkedList() {
  super();
  tako.lang.TKObject $this = (tako.lang.TKObject)(this);
  tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
  tkvariables.func[0] = (java.lang.Object)(new Node());
  this.head = ((Node)tkvariables.func[0]);
  this.pre = this.head;
  this.last = this.head;
  tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
}

public void insert(tako.lang.TKObject x) {
  tako.lang.TKObject $this = (tako.lang.TKObject)(this);
  java.lang.Object result = null;
  tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
  Node post = (Node)(null);
  Node new_pos = (Node)(null);
  post = this.pre.next;
  tkvariables.func[0] = (java.lang.Object)(new Node());
  new_pos = ((Node)tkvariables.func[0]);
  tako.lang.TKObject.swap(new_pos.contents, x);
  if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[1]))
    new_pos.contents = ((tako.lang.TKObject)tako.lang.TKGlobals.params[1]);
  else new_pos.contents = new tako.lang.TKObject();

  if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[2]))
    x = ((tako.lang.TKObject)tako.lang.TKGlobals.params[2]);
  else x = new tako.lang.TKObject();
  this.pre.next = new_pos;
  new_pos.next = post;
  if (this.right_length == 0) {
    this.last = this.last.next;
  }
  this.right_length++;
  tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
  tako.lang.TKGlobals.params[1] = (java.lang.Object)(x);
}

public void remove(tako.lang.TKObject x) {
  tako.lang.TKObject $this = (tako.lang.TKObject)(this);
  java.lang.Object result = null;
  tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
  Node old_pos = (Node)(null);
  Node new_post = (Node)(null);
  new_post = this.pre.next;
  old_pos = new_post;
  new_post = new_post.next;
  this.pre.next = new_post;
  tako.lang.TKObject.swap(x, old_pos.contents);
  if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[1]))
    x = ((tako.lang.TKObject)tako.lang.TKGlobals.params[1]);
```

```
  else x = new tako.lang.TKObject();

  if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[2]))
    old_pos.contents = ((tako.lang.TKObject)tako.lang.TKGlobals.params[2]);
  else old_pos.contents = new tako.lang.TKObject();

  if (this.right_length == 1) {
    this.last = this.pre;
  }
  this.right_length--;
  tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
  tako.lang.TKGlobals.params[1] = (java.lang.Object)(x);
}

public void advance() {
  tako.lang.TKObject $this = (tako.lang.TKObject)(this);
  java.lang.Object result = null;
  tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
  this.pre = this.pre.next;
  this.left_length++;
  this.right_length--;
  tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
}

public void reset() {
  tako.lang.TKObject $this = (tako.lang.TKObject)(this);
  java.lang.Object result = null;
  tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
  this.pre = this.head;
  this.right_length = (this.left_length + this.right_length);
  this.left_length = 0;
  tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
}

public void get(tako.lang.TKObject out) {
  tako.lang.TKObject $this = (tako.lang.TKObject)(this);
  java.lang.Object result = null;
  tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
  if (this.pre.next != null) {
    tako.lang.TKObject.swap(out, this.pre.next.contents);
    if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[1]))
      out = ((tako.lang.TKObject)tako.lang.TKGlobals.params[1]);
    else out = new tako.lang.TKObject();

    if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[2]))
      this.pre.next.contents = ((tako.lang.TKObject)tako.lang.TKGlobals.params[2]);
    else this.pre.next.contents = new tako.lang.TKObject();
  } else {
    java.lang.System.out.println("pre.next is null");
  }
  tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
  tako.lang.TKGlobals.params[1] = (java.lang.Object)(out);
}

public int length() {
  tako.lang.TKObject $this = (tako.lang.TKObject)(this);
  int int_result = 0;
```

```
    tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
    int_result = (this.left_length + this.right_length);
    tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
    tako.lang.TKGlobals.int_result = int_result;
    return ((int)tako.lang.TKGlobals.int_result);
  }

  public boolean isEmpty() {
    tako.lang.TKObject $this = (tako.lang.TKObject)(this);
    boolean boolean_result = false;
    tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
    boolean_result = (this.left_length + this.right_length) == 0;
    tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
    tako.lang.TKGlobals.boolean_result = boolean_result;
    return ((boolean)tako.lang.TKGlobals.boolean_result);
  }
}
```

Hash Set (Tako code)

```
public class HashSet {

  private static final int DEFAULT_CAPACITY := 11;

  private static final float DEFAULT_LOAD_FACTOR := 0.75f;

  private int size;

  private int threshold;

  private LinkedList[] buckets;

  public HashSet() {
    size := 0;
    threshold := (int) (DEFAULT_CAPACITY * DEFAULT_LOAD_FACTOR);
    buckets := new LinkedList[DEFAULT_CAPACITY];

    for (int i := 0; i < buckets.length; i++) {
      buckets[i] := new LinkedList();
    }
  }

  public void insert(TKObject e) {
    //assert !contains(e);
    int idx := hash(e);
    buckets[idx].insert(e);
    size := size + 1;
  }

  public void removeAny(TKObject e) {
    //assert !isEmpty();
    int count := 0;
    while (buckets[count].isEmpty()) {
      count++;
    }
    size := size - 1;
    buckets[count].reset();
    buckets[count].remove(e);
  }

  public int size() {
    result := size;
  }

  private int hash(TKObject e) {
    result := (e = null ? 0 : Math.abs(e.hashCode() % buckets.length));
  }
}
```

Hash Set (Java translation)

```
public class HashSet extends tako.lang.TKObject {

  private static final int DEFAULT_CAPACITY = 11;
  private static final float DEFAULT_LOAD_FACTOR = 0.75F;
  private int size;
  private int threshold;
  private adventure.util.LinkedList[] buckets = null;

  public HashSet() {
    super();
    tako.lang.TKObject $this = (tako.lang.TKObject)(this);
    tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
    this.size = 0;
    this.threshold = 8;
    this.buckets = new adventure.util.LinkedList[11];

    for (int i = 0; i < this.buckets.length; i++) {
        tkvariables.func[0] = (java.lang.Object)(new adventure.util.LinkedList());
        this.buckets[i] = ((adventure.util.LinkedList)tkvariables.func[0]);
    }
    tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
  }

  public void insert(tako.lang.TKObject e) {
    tako.lang.TKObject $this = (tako.lang.TKObject)(this);
    java.lang.Object result = null;
    tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
    int idx;
    this.hash(e);
    $this =
        (tako.lang.TKObject)(((adventure.util.HashSet)tako.lang.TKGlobals.params[0]));
    if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[1]))
      e = ((tako.lang.TKObject)tako.lang.TKGlobals.params[1]);
    else e = new tako.lang.TKObject();
    idx = ((int)tako.lang.TKGlobals.int_result);
    this.buckets[idx].insert(e);
    this.buckets[idx] = ((adventure.util.LinkedList)tako.lang.TKGlobals.params[0]);
    if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[1]))
      e = ((tako.lang.TKObject)tako.lang.TKGlobals.params[1]);
    else e = new tako.lang.TKObject();
    this.size = (this.size + 1);
    tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
    tako.lang.TKGlobals.params[1] = (java.lang.Object)(e);
  }

  public void removeAny(tako.lang.TKObject e) {
    tako.lang.TKObject $this = (tako.lang.TKObject)(this);
    java.lang.Object result = null;
    tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
    int count = 0;

    while (this.buckets[count].isEmpty()) {
      count++;
```

```
    }
    this.size = (this.size - 1);

    this.buckets[count].reset();
    this.buckets[count] = ((adventure.util.LinkedList)tako.lang.TKGlobals.params[0]);
    this.buckets[count].remove(e);
    this.buckets[count] = ((adventure.util.LinkedList)tako.lang.TKGlobals.params[0]);

    if (tako.lang.TKObject.class.isInstance(tako.lang.TKGlobals.params[1]))
      e = ((tako.lang.TKObject)tako.lang.TKGlobals.params[1]);
    else e = new tako.lang.TKObject();
    tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
    tako.lang.TKGlobals.params[1] = (java.lang.Object)(e);
  }

  public int size() {
    tako.lang.TKObject $this = (tako.lang.TKObject)(this);
    int int_result = 0;
    tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
    int_result = this.size;
    tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
    tako.lang.TKGlobals.int_result = int_result;
    return ((int)tako.lang.TKGlobals.int_result);
  }

  private int hash(tako.lang.TKObject e) {
    tako.lang.TKObject $this = (tako.lang.TKObject)(this);
    int int_result = 0;
    tako.lang.TKVariables tkvariables = new tako.lang.TKVariables();
    tkvariables.int_func[0] = e.hashCode();
    tkvariables.int_func[1] = java.lang.Math.abs((tkvariables.int_func[0] %
                                                  this.buckets.length));
    int_result = e == null ? 0 : tkvariables.int_func[1];
    tako.lang.TKGlobals.params[0] = (java.lang.Object)($this);
    tako.lang.TKGlobals.params[1] = (java.lang.Object)(e);
    tako.lang.TKGlobals.int_result = int_result;
    return ((int)tako.lang.TKGlobals.int_result);
  }
}
```