SOURCE-LEVEL REWRITING OF THE JUMBO RUNTIME CODE GENERATION FRAMEWORK

BY

PHILIP MORTON

B.S., University of Nebraska at Omaha, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

Jumbo is a run-time code generation framework for Java. Jumbo programs are composed of code fragments; when those fragments appear in a static context, some of the dynamic checks in the compiler may become unnecessary. A source-level rewriting system has been used to optimize Jumbo-created code generators with the intent to remove this inefficiency. We detail the problems encountered in using this rewriting system, and describe our efforts to improve it, including the addition of an alias analysis and rewriting rules to take advantage of alias information. We illustrate the effect of these additions with an example.

Table of Contents

Chapter

Chapter 1

Introduction

1.1 Basis for this work

Jumbo is a run-time code generator for Java. The Jumbo framework allows programmers to write programs that generate and combine fragments of other programs, culminating effectively in the run-time compilation of new classes.

Of course, compilation of classes at run-time exacts considerable overhead. Therefore, we have been looking into ways to optimize code generators written with Jumbo. The focus of our efforts so far has been a source-level rewriting system, originally written by Lars Clausen [1].

1.1.1 Jumbo

Jumbo works with an extended Java syntax allowing quoted "code fragments" to represent code that is to be dynamically generated. This eases the process of writing code generators with Jumbo; the dynamic code fragments are easy to read and recognizably Java.

The extended syntax is not required, however. At static compilation time, the Jumbo parser converts the program into valid normal Java. This involves replacing all the quoted code segments with calls to the Jumbo API. At runtime, these calls will create numerous

Code objects representing AST fragments of the formerly quoted code. These Code objects can then be passed to other methods, or serialized for later consumption, and used to fill in "holes" in the ASTs of other code sections. When code representing a complete class has been filled in, calling the "generate" method on the Code object will begin compilation. After several passes, the classfile is written to disk, and the generated class can be loaded and used like any other.

This expansion of quoted syntax into regular Java often results in inefficient code. Each piece of quoted syntax is expanded independently, so the expanded code can contain many checks for conditions which are determinable from the context. This inefficiency can exist even in a small fragment, or in quoted code which contains holes, although (in our experiences optimizing the code) it is most prevalent when a quoted section contains a complete classfile.

The Jumbo rewriting system is designed to remove this inefficiency through source-level optimizations. The quoted syntax of a generator is expanded to regular Java, and the rewriting system is used to optimize the expanded program in a series of automatic and manual steps. The rewriting system attempts to resolve any dynamic checks which can be statically determined given the context of the code fragment.

The rewriting system is not specific to Jumbo; these rewriting rules could as easily be applied to any standard Java program. The design of the rewriting system has been driven by the need to optimize Jumbo code, however, and so our extensions of the rewriting system have focused on solving the problems encountered in that domain.

The first section is devoted to describing the rewriting system as it existed before our efforts. Next, we describe some of the problems we encountered while attempting to use the older rewriting system. Then we discuss the additions we have made to the rewriting system in an attempt to resolve those problems. Our alias analysis receives a separate chapter. Finally, we illustrate the use of the new additions with an example, and conclude with our observations on the effectiveness of our additions.

Chapter 2

Basis

2.1 The Rewriting System

First, we will explain the analyses used in the rewriting system (other than the alias analysis).

The rewriting system uses use-def, flow, and dominator analyses to generate information needed for the rewriters. Use-def and flow analyses are run before each application of any rewriter; dominator information is used by only a few rewriters, and so is only run before attempting to apply those rewrite rules.

We will now explain the existing rewriters.

2.1.1 Preexisting rewriters

The following rewriters existed have been changed minimally since [1].

2.1.1.1 UnusedDecl

This rewriter removes variable declarations that the use-def analysis indicates are not used in the method.

2.1.1.2 UnusedScope

This removes scopes that have no significance. There must be no break statements changing the control flow and no variable name shadowing (declarations of variables with the same names as variables in an outer scope) in the scope to be removed.

2.1.1.3 UnusedDef

This rewriter removes definitions that the use-def analysis indicates are not used.

2.1.1.4 IfReduction

This rewriter reduces if-statements when the branch condition is a constant. It has been improved to also reduce if-statements where both branches are identical.

> *This situation generally occurred after other rewriters had reduced both branches of an if-statement. More specifically, it happened often in the code that checks the legality of flags for a class or method. This code branches depending on whether the code in question is inside an inner class, but for some flag values (public non-static, for example) the result will be the same on both branches. The improvement to IfReduction allows this situation to be resolved even when we cannot determine our context.*

2.1.1.5 Arithmetic

Resolves constant-valued expressions.

2.1.1.6 UnusedBreak

Removes break statements that do not affect the flow: those that do not break out of loops or past any statements.

2.1.1.7 Untupling

The first of two rewriters that could propagate information through fields. Untupling requires that the target of the field access be unambiguously traceable to a new object construction, and the field in question must be directly assigned exactly once in the constructor. There are also restrictions on the uses of the object, if the field is non-final.

2.1.1.8 FieldValue

The second rewriter that could propagate information through fields. This rewriter differs from Untupling in that it will attempt to trace back further to find the target, but it can only handle final fields. It shares with Untupling the restriction that the field must be directly assigned inside the constructor, and must be assigned a parameter or literal.

2.1.1.9 ConstantPropagation

Utilizes use-def information to propagate constants through local variables.

2.1.1.10 CollapseSystemCalls

This rewriter reduces some known system calls (`String.equals` when the target and argument are string literals, for example).

2.1.1.11 ArrayLength

Attempts to resolve the `array.length` field lookup on arrays, if possible. This rewriter and the CollapseSystemCalls rewriter are rather specialized, but useful for resolving issues that the more general-purpose rewriters cannot; `String.equals` calls are often used as conditions for branches which we would like to eliminate, and `array.length` values are involved in loop conditions.

2.1.1.12 Switch

Resolves switch statements if the condition is a constant.

2.1.1.13 CopyAssignment

Attempts to move definitions (assignments) of variables to their uses, if the right hand side of the assignment was another variable and it would not have been changed between the definition and the use.

2.1.1.14 UnusedReturn

Removes assignments of the return value of method or constructor calls, if those return values are never used.

2.1.1.15 UnusedObject

Removes constructor calls if they have no side effects and the new object is never used.

2.1.1.16 TightenType

Attempts to reduce variable types from supertypes to subtypes, if we can see that the variable will be used to hold only objects of a subtype.

2.1.1.17 UnusedFieldAssign

Removes field assignments that are never used.

> *Initially, the restriction was that the target object must be used only in field accesses other than field lookups on the assigned field and must not escape its constructor. We expanded this by using alias information; if the target has no aliases at the point of the field assignment, then any use of the target that is not reachable from the assignment is safe.*

2.1.1.18 SingleUse

This rewriter is used to reverse flattening. It moves a definition with a single use to that use.

2.1.1.19 WhileIf

Reverses the flattening of loops by moving `if(condition) break;` statements in the body into the condition of the loop.

2.1.2 Manual rewriters

The following rewriters are not automated, and must be activated manually.

2.1.2.1 Inlining

A manually-activated rewriter that attempts to inline a method call. It has been upgraded to use must-alias information to locate the method, if available.

> *There have also been a few additions to ensure that the behavior of inlined functions is identical to their behavior before inlining: namely, ensuring that the target of the method call is cast to a reference of the type of the class in which the method was defined, and ensuring that a null pointer exception is thrown if the target of the method call is null. The first change was in response to problems encountered while rewriting; if the method to be inlined was located by a means other than looking at type information, using the target without casting could lead to type problems. An example would be when a method was called on a variable of an interface type; if use-def information or alias information was used to find the true type of the object to locate the method, and the method included field accesses on the target, inlining the method without casting the target would result in field accesses on a variable of the interface type. The second change was not in response to any encountered problem, but to the potential problem that someone could write a program relying on the JVM to throw a NullPointerException when a method was called on a null target, and the original inlining code could break that functionality: if the class of the target variable contained a* `final` *method with the appropriate*

name and signature, the Inlining rewriter would allow the method to be inlined even if the variable could be null.

2.1.2.2 WhileUnroll

Unrolls a while loop once. This is another rewriter which must be applied manually.

2.1.3 New rewriters

The next seven rewriters are recent.

2.1.3.1 ObjectEquality

Attempts to resolve equality and inequality comparisons between objects.

2.1.3.2 NullCheck

Attempts to resolve equality and inequality comparisons with null.

2.1.3.3 InstanceOf

Attempts to resolve `instanceof` expressions.

2.1.3.4 AnonClassConvert

A rewriter that converts anonymous classes to named inner classes.

This rewriter was implemented to give us a means to reduce the size of rewritten methods, which was causing us difficulty by slowing down our analyses. We also had some hope from the results of [2] that specializing named inner classes could give us more improvement.
Unfortunately, this rewriter has not gotten much use. The specialization of inner classes in Mumbo was valid because the methods there had no side effects, a claim which we could not make about the methods of Jumbo, even after

restructuring it to make it more amenable to optimization. This meant that our ability to specialize `Code` *classes comes only from the fact that we can specialize anonymous classes using information about their context. Without anonymous classes, we would only be able to optimize generators in the presence of a* `generate` *or* `create` *call, which would drastically cut down the number of cases in which optimization was possible.*

2.1.3.5 WhileReduction

This rewriter removes while loops (or other blocks) where the first statement is a `break` out of the loop or block (but no further).

2.1.3.6 PointlessCast

This rewriter removes casts where the target of the cast is already of the appropriate type.

2.1.3.7 UnreachableCode

This rewriter removes code that cannot be reached from the entry of the method.

Occasionally, it is necessary to run this rewriter after other rewriters in order to obtain valid code. Specifically, the use-def analysis works on the control flow graph, and so will miss the uses of a variable if those uses are in an unreachable location. Rewriters such as UnusedDef and UnusedDecl then remove the variable definitions and declarations, with the result that the unreachable code contains references to variables that no longer exist. Running UnreachableCode resolves the problem.

2.1.4 Alias and temporary constructor inlining rewriters

The remaining rewriters concern the alias analysis and temporary constructor inlining, which comprise the bulk of this thesis.

2.1.4.1 ConstructorInlining

This manually activated rewriter inlines a constructor, according to the principles of our temporary constructor inlining. It will be described further in section {something}.

2.1.4.2 ConstructorAnnotationRemove

Removes the temporary constructor annotations. This is mostly used to improve the performance of the rewriters by reducing the complexity of the abstract syntax tree.

2.1.4.3 AliasCheck

This rewriter runs the may-alias analysis on the program. It does not change the abstract syntax tree, but provides information that may be used by other rewriters.

2.1.4.4 MustAliasCheck

This rewriter runs the may-alias analysis, then the must-alias analysis. Like the AliasCheck rewriter, it does not change the abstract syntax tree, but other rewriters may use the information it provides.

2.1.4.5 AliasRemove

This removes alias information from the program. Like ConstructorAnnotationRemove, its only purpose is to speed up the rewriting process by freeing the memory that was being used to hold the alias information.

2.1.4.6 AliasTraceRewriter

This rewriter is primarily for debugging and manual analysis purposes. The size of the rewritten programs and the space complexity of the alias information, combined with the clumsiness of the user interface (especially when system memory is low), makes it very difficult to determine (for example) at what point in the program the alias analysis claims that a given name escaped. Locating this point often illuminates aspects of the

program's behavior that were not obvious or were overlooked in planning the initial strategy for rewriting. Invoking this rewriter allows the user to reduce the amount of alias information being displayed, so that only the information pertinent to the alias in question will be visible.

2.1.4.7 AliasConstantRewriter

This is the first proper (AST-changing) rewriter that is strictly dependent on alias information. It uses must-alias information to replace constant-valued variables and field accesses with their values. It fulfills the same purpose as ConstantPropagation, but is more effective because it is not limited to local variables. For more information on how the must-alias analysis keeps track of constants, see section 4.1.2.

2.1.4.8 AliasObjectEqualityRewriter

This rewriter fulfills the same purpose as the ObjectEquality and NullCheck rewriters, but uses must-alias information to do it. It is more effective than ObjectEquality or NullCheck at resolving comparisons when they involve information from field accesses.

2.1.4.9 AliasPropagateRewriter

Both this rewriter and the AliasFieldReplacementRewriter were designed with the idea of implementing functionality similar to CopyAssignment, using the must-alias analysis information. The idea was that if the must-alias analysis tells us that two names are aliased at a point, then it must be safe (modulo scoping issues) to replace one name with the other at that point. The problem, however, is that there may be several possible names suitable for this replacement, and with no obvious condition making one name more useful than another, we have no obvious termination condition for the rewriter.

The first idea was to use the must-alias analysis' ability to keep track of constant names for objects to determine which names should get priority in a replacement. If the must-alias analysis can identify a particular object that a name is pointing to, then all names must-aliased with the given name must point to that same object; if we can

associate a particular name more strongly with the object than the others, we would have a condition for choosing a best replacement. Therefore, we attempt to replace a name with a known alias to an object with the variable to which the object was assigned immediately after its creation. We use the same test as in CopyAssignment to ensure that this replacement is valid, because the original name may have been reassigned or gone out of scope.

Requiring a constant object alias to be known for each alias to be replaced proved to be restrictive, however; worse, this method of choosing a name for the replacement ignored the idea driving the creation of this rewriter in the first place, which was to utilize the principle that it was safe to replace names with other names that were must-aliased with them. There is no guarantee that the name first assigned to the object creation will be must-aliased with the name we want to replace. This rewriter was eventually replaced by the AliasFieldReplacementRewriter.

2.1.4.10 AliasFieldReplacementRewriter

In looking at the cases where we wanted to use the AliasPropagate rewriter but were unable to (because we lacked a visible object creation or the first variable to which the object was assigned was not propagateable), we noticed that in most of the situations where we wanted to replace names with aliases, the names to be replaced were field accesses. While our alias analysis was very good (compared to the rest of the rewriting system) at tracing information through field accesses, taking advantage of that in every other rewriter would have required updating them all to interface directly with the alias information; we had modified several rewriters to do that already, but it was a difficult and somewhat risky prospect. Some of the modified rewriters had already caused validity problems when the use of alias information to loosen their restrictions violated assumptions made in other areas of the rewriter. Therefore, it made sense to separate the concerns and use a single alias-aware rewriter to translate the field accesses that troubled the other rewriters into uses of local variables, if possible. This provided a rule for replacing names using alias information–if we replaced field accesses with local

variables, the rewriter would be bounded and we would be certain that every application of the rewriter reduced the complexity of the program.

The AliasFieldReplacement rewriter, therefore, replaces field accesses with must-aliased non-field-access names, if those names are in scope at the point of replacement. The replacement names are chosen arbitrarily; the first in-scope non-field-access alias name found is used as a replacement. It is possible that a smarter selection of replacements could give us more information, but the existing rewriters and the use-def analysis seem to be generally good at handling local variables.

Chapter 3

Rewriting Problems

We encountered numerous problems while trying to apply rewrites to code generators. Often, there would be an area where we would expect to be able to simplify, but it would be blocked because we did not have rewrite rules that could handle that situation. Other times, we could see through inspection that a value was safe to propagate, but were unable to prove it in such a way that a generic rewrite rule could deal with the operation.

3.1 Propagating values through fields

This was one of the most prevalent "We have the information but we can't use it," problems. The two preexisting rewriters dealing with this situation were limited, only dealing with final fields and able to determine only cursory information from the constructors of the objects they were examining.

This left many places where fields should have been statically determinable, but were not. In particular, information about the compilation environment, like the current class being worked on, are stored in an environment class. This information is used widely in compilation but could not be resolved under the old system. For example, even when the environment was newly created, the current class field would be initialized through a superconstructor call, which none of the existing rewriters could follow.

3.2 Rewriting for future execution

Jumbo is a framework for staged compilation. Quoted code fragments will not be compiled until they are part of a complete class, and indeed may never be compiled at all (the code fragments could be consumed by some generator which picks and chooses a few fragments for compilation from a wide selection). The code generators themselves, therefore, just create Code objects with methods that will perform compilation when called.

There is limited opportunity for optimization on the creation of these Code objects; most of the processing takes place when `generate` is called. Unfortunately, we do not always have access to the call to `generate`; one of the central expectations for Jumbo is that it will be used to bring together code fragments from disparate sources. This suggests that many uses of Jumbo will be to generate and return Code objects; optimizing these uses gives us little opportunity to optimize the actual compilation of the fragment, which is where the real cost occurs.

Several things keep the rewriting system useful, however.

First, not all uses of Jumbo require code fragments from different sources. Specializing on vectors in dot products (one of the common examples of the usefulness of staged compilation) can be done by generating the code and compiling it in the same module, with the result that the client of the specializer does not even have to know that code generation is being used. In situations like this, the compilation can be optimized directly. This suggests that there may be external factors encouraging code generators to be written in a form amenable to rewriting.

Second, even when fragments from multiple sources are being combined, compilation has to occur somewhere. In several examples we've seen of combining code fragments, the fragments are placed in a simple holder class before compilation. Even without knowledge of the contents of the incoming fragments, there are many checks in the compilation that can be simplified with knowledge of the containing class (determining whether the enclosing class is an inner class, for example).

Finally, we are not lost even if the code we wish to optimize does no compilation at all, but we are much more limited. For the most part, the Code objects are implemented as anonymous classes. After Jumbo converts quoted code fragments into calls to functions that return Code objects, we can inline those calls to create new syntactic copies of the anonymous classes that would be returned. The upside of using anonymous classes is that it exposes the methods that will be called during compilation; we can use the rewriting system to specialize these methods, effectively speeding up compilation even though we do not know when or where compilation will occur. This is what is meant by "rewriting for future execution."

Rewriting the methods of anonymous classes still has some problems, however. Each pass of the compiler is represented by a different method; optimizations are much more difficult when we do not have the context in which the methods will be called, and it is almost impossible to propagate information across methods.

3.3 Size of rewritten code

Rewriting the code involves inlining a lot of functions, including the aforementioned functions that create Code objects; as the Code objects are implemented as anonymous classes, inlining these functions results in the duplication of large blocks of code. This improves our ability to specialize, but results in vastly increased code size. In addition to slowing down the rewrite system itself, this can cause slowdown at runtime when a much-larger classfile needs to be loaded. There is also some evidence that larger methods are optimized less aggressively by the Java Hotspot compiler .

3.4 Dealing with "holes" in the code

Code fragments do not need to be complete, although they cannot be compiled until they are. Holes in the generated code manifest themselves as calls to the compilation methods of Code objects obtained from some untraceable source (passed in as parameters, read

from streams, etc...) These cause difficulty when rewriting because we cannot predict their behavior.

For example, during compilation there will be an object representing the current compilation environment. This object could contain some non-`final` fields, such as a field keeping track of the current class for which code is being compiled. If the code fragment contains a class declaration, then there will be a section of code setting the value of this field to the class in the quoted fragment.

If a hole then appears inside the quoted class, we run the risk that compiling the unknown code provided for the hole (i.e., calling the compilation methods on the untraceable Code object) will cause a change to the "current class" field of the environment. Even when no known Code object has a method that would cause a change like that, we cannot rule it out because a user could, potentially, create a new subclass of Code with behavior that breaks that assumption. This then causes difficulty when we wish to propagate information past the hole; we determine that the value of the current class field (for example) depends on the behavior of the hole, which is unknown, and so optimization of code after the hole must proceed without knowledge of the current class.

Of course, if we knew that no current subclass of Code would need to change a given field of the environment, and could see no reason for a user to manually subclass Code to change that field, then making the field `final` would be the best way to resolve this problem. This was the focus of the restructuring of the compiler seen in [3].

Chapter 4

Additions

4.1 Additions to the Rewriting System

We made several modifications and additions to the rewriting system in order to enable additional optimizations.

4.1.1 New standalone rewriters

In some instances, potential simple optimizations were blocked only by the fact that we did not have rewrite rules to take advantage of them. We created new rewriters for these cases.

4.1.1.1 PointlessCast

This rewriter removes cast expressions when it can determine that the target of the cast is already of the appropriate type.

4.1.1.2 UnreachableCode

This rewriter removes code that cannot be reached. This is necessary to maintain the validity of some of the other rewriters. The use-def analysis uses flow information to determine whether a variable is used or not. If the only use of a variable occurs in a

block that is unreachable by any control path, the use-def analysis will not record it. This can lead to the UnusedDef and UnusedDecl rewriters removing the definition and declaration of the variable, resulting in invalid code: the unreachable code contains a use of a variable that was never declared. Removing all code that is unreachable on any control flow path will fix this issue.

4.1.2 Temporary Constructor Inlining

As mentioned above, one of the more common situations we ran into was that we would find places where computation should only depend on static information, but we could not propagate that information to the necessary site. Often, the information was "lost" when it was stored in an object (such as the Environment) and later retrieved.

For example, there are many places during compilation where the current class needs to be known. The current class is stored as a field in the environment. This field is set when we start evaluating the class, and is propagated through the rest of the compilation as more environments are created. Several methods of compilation create their own environments after being called, with the current class field copied from the incoming environment.

This means that when we want to look up the currentClass field from an environment, we have to locate the creation of that environment, examine its constructor, and determine where the currentClass field came from. If it turns out that it came from a field of one of the parameters to the constructor, we have to locate *that* object's creation, look at *its* constructor, etc...until we find a definition we can use (an object creation of the ClassInfo object residing in the currentClass field, an assignment of `null` to the field, or a variable that we can propagate). Complicating this was the fact that we could not specialize the constructors; with no way to inline constructors, we could not propagate information from the constructor arguments to the body and so could not resolve conditionals or method calls. We could only trace the field assignment through a very simple constructor. Control flow or method calls inside the constructor could prevent us from determining where the field was assigned, or what value was assigned to it.

Temporary constructor inlining was designed as a way to allow us to specialize the constructors, letting us use existing rewriters to deal with complicated statements inside the constructor and revealing the field assignments we need. It does this by adding annotations, containing the statements of the constructor, to object creation sites. Other rewriters then see the constructor code as though it were just an inlined method. There is no valid Java syntax for specializing a constructor, so the annotations must be removed before the optimized program is written; for this reason, the annotations must have the property that they can be removed at any time and leave a program with the same meaning as when they were there.

4.1.2.1 Why is it valid?

The intuition for why temporary constructor inlining is valid is that, essentially, an object creation can be seen as an allocation of memory followed by a call to an initialization method. By treating the constructor call of an inlined constructor as an allocation of memory, we preserve that meaning in a reversible fashion. Object creations that have not been inlined are combination allocation/initialization calls, and object creations that have been inlined are memory allocations followed by (inlined) initialization calls.

The assignment of the new object to an identifier does become out of place after this transformation: in a non-inlined object creation, that assignment would not happen until after initialization, while in our inlined creations it comes before. Therefore, we use a fresh variable to create the object and assign it to the original target after initialization is complete. When the annotations are removed, this assignment through a temporary variable becomes essentially a no-op, so the meaning of the program is preserved. (One downside of introducing this fresh variable is that the CopyAssignment rewriter seems to give precedence to the newer variable over the old one when eliminating the assignment; since fresh name generation is currently limited to appending generated numbers onto the old name, this causes the names of objects that are repeatedly the subjects of temporary constructor inlining to be monotonically increasing. A better unique name generation system could solve this.)

Chapter 5

Alias Analysis

The partition-based alias analysis system stores alias information as partitions of the names in a program. Names in the same partition may be aliases of each other at run-time. The analysis is flow-sensitive and context-insensitive.

For example, the partitioning `[a, b] [b.f, c]` would represent that `a` and `b` may be aliased (or must be aliased, if that partitioning was a result from the must-alias analysis), and that the `.f` field of `b` (and `a`, by implication) could be aliased with `c`.

If the above partitioning occurred at the point before the statement `a = c`, then the partitioning after the statement would be `[b] [ b.f, c, a]`.

5.1 Text description of the algorithm

5.1.1 May-alias

In the may-alias analysis, we start with an environment that contains one partition: the Dump (which contains the special name "D*" and represents names which can point to objects that have escaped the current method). At the start of each method, the Dump also contains the parameters of the function and the name "this". We also have names *D*.f* in the dump at initialization for every field *f* in the program.

We then do a standard dataflow analysis, performing joins and transfers as below; this will be formalized in section 5.2. The may-alias analysis has two different types of transfers: *move* and *merge*. The first represents a transfer where it is safe to kill old aliases; the second is a transfer in which old aliases must be preserved. Both are parameterized on the names to be affected and the incoming alias information; $move(E, T, target)$ takes the set of names whose aliases should be killed (T) and the name to which they should have new aliases created ($target$). $merge(E, S)$ takes all of the names which should now be aliased. E in both cases represents the inflow data: the partitioning of names representing the alias information at the program point preceding this statement. Generally, the only time it is safe for us to kill old may-aliases is when we have a simple variable on the left hand side of an assignment.

5.1.1.1 x = y

The transfer function when x is a simple name is $E_{next} = move(E, \{x\}, y)$. In an assignment to a simple name, unlike the cases below, it is always safe to remove the name from its partition because there is no ambiguity about whether another name is affected by the assignment.

If x is a field access, then $E_{next} = merge(E, \{x, y\})$ instead. We cannot kill aliases of field accesses in the may-alias analysis because the aliases of the target of the field access may or may not represent actual aliases at runtime. If a may be aliased with b, an assignment to $a.f$ may or may not change the value of $b.f$, depending on whether a and b are truly aliased at runtime. The analysis must therefore reflect that $b.f$ may be aliased with either the value assigned to $a.f$ or the value held previously by $b.f$. In our representation, the only way to do this is to merge the partitions of $a.f$ and y.

If x is an array access `x[i]`, $E_{next} = merge(E, \{x.\_element, y\}$; similarly, if y is an array access `y[i]`, $y.\_element$ replaces y in any of the above. We do not attempt to model arrays; using a unique field name to represent any element of an array captures all possible aliases of any element of the array, because aliases created by an assignment

to or from any element of the array will not be killed unless the underlying array is reassigned.

5.1.1.2 x = (class)y

The effect of this statement is the same as the effect of `x = y`. We do not incorporate type information in the alias analysis, and so casting is not an interesting operation for us.

5.1.1.3 x = c

If the right hand side is a primitive constant c, and x is not a field or array access, $E_{next} = move(E, \{x\}, \emptyset)$. Assigning the name to a primitive constant means that the name is not aliased with any other name.

If x is a field or array access, $E_{next} = E$.

5.1.1.4 x = new c($y_1, y_2 ... y_n$)

If the constructor call has been inlined, $E_{next} = move(E, \{x\}, \emptyset)$, unless x is a field or array access, in which case $E_{next} = E$. An inlined constructor call does not represent an escaped object unless the code of the constructor indicates an actual escape of the new object or any of the parameters; assignment of the real arguments to the formal parameters is made explicit by the constructor inlining, and so does not need to be handled here. If the assignee is not a field access, we can kill its aliases because we know the object is fresh (aliases created in an inlined constructor will be explicit); if the assignee is a name whose aliases we should not kill, then this call has no effect.

If the constructor call has not been inlined, we need to do two transfers here: $E_{next} = move(merge(E, \{y_1, y_2, ...y_n, D*\}), \{x\}, D*)$. The effects here are the same as a standard method call; the parameters escape and x is assigned to an object that could have escaped. If x was a field or array access, however, we cannot kill its aliases; in that case, $E_{nexT} = merge(E, \{y_1, y_2, ...y_n, x, D*\})$.

5.1.1.5 x = method($y_1, y_2, ...y_n$)

$E_{next} = move(merge(E, \{y_1, y_2, ...y_n, D*\}), \{x\}, D*)$, if x is a simple name. If x was a field or array access, then $E_{nexT} = merge(E, \{y_1, y_2, ...y_n, x, D*\})$.

See the non-inlined constructor call for the reasoning behind this equation.

5.1.1.6 Inner class declaration

First, determine all names that appear inside the class declaration and also inside E. Let this set of names be called N. $E_{next} = merge(E, N \cup \{D*\})$

This represents that all free names used in the inner class could escape and be used in ways unknown to us. It also parallels the way inner classes would be handled if the syntactic sugar were removed (by converting them to ordinary non-inner classes); all free variables become arguments to the constructor. (The object of the outer class, if in a non-static context, would also escape in this way, but this does not require any action on the part of our analysis because we consider the name *this* to be inherently escaped.)

5.1.2 Must-alias

The must-alias analysis does not use *merge*; killing aliases is the default correct behavior, so there is no need for an alias-preserving transfer function. Must-alias analysis introduces a transfer called *lossOfControl*(E), which represents the effects that unknown methods can have on escaped objects when the control flow departs from the code which we are analyzing. An unknown method could potentially change any non-final field on any object to which it has access; we use the may-alias information to determine which objects could be accessed by unknown methods at any point (those objects which have escaped to the Dump).

5.1.2.1 x = y

$E_{next} = move(E, \{x\}, y)$

This is simple enough. Unless the name is an array access, we know for certain that the alias created by this assignment will hold immediately after this statement, so the above transfer holds as long as x and y are either simple names or field accesses.

If x is an array access, do nothing. If y is an array access, $E_{next} = move(E, \{x\}, \emptyset)$. We do not keep track of any must-alias information about array elements.

5.1.2.2 x = (class)y

$E_{next} = move(E, \{x\}, y)$

If x is an array access, do nothing. If y is an array access, $E_{next} = move(E, \{x\}, \emptyset)$.

Again, we ignore casting.

5.1.2.3 x = new c($y_1, y_2, ...y_n$)

$E_{next} = move(E, \{x\}, constant(\texttt{new c(...)}))$ if the constructor call has been inlined. The *constant* constructor creates a special name representing the object created at a constructor call; this is not strictly necessary for this analysis, but it is useful for several applications and can allow for a more precise analysis by giving us type information which we can use to determine field finality.

Otherwise, $E_{next} = move(lossOfControl(E), \{x\}, constant(newc(...)))$

If the constructor call has not been inlined, then it represents a loss of control which we have to process. Either way, we know that x is now aliased with the new object (represented by the special Constant name).

5.1.2.4 x = c

If the right hand side is a primitive constant c, $E_{next} = move(E, \{x\}, constant(\texttt{c}))$.

5.1.2.5 x = method($y_1, y_2, ...y_n$)

$E_{next} = move(lossOfControl(E), \{x\}, \emptyset)$

We handle the loss of control, and kill the aliases we had for x.

5.1.2.6 Inner class declaration

If this inner class declaration is an anonymous class creation, we handle a loss of control ($E_{next} = lossOfControl(E)$; arguably, this is covered under the "Constructor" case). Otherwise, we do nothing.

5.2 Formal rules

We now formalize the alias analysis rules.

5.2.1 May-alias

5.2.1.1 Join

The join function here constructs a partitioning in which any two names which were in the same partition in any incoming partitioning are in the same partition in the joined partitioning.

$join(P_1, P_2, ...P_n) =$
$\{\{x_i \mid \exists P_1...P_{i-1} :$
$P_1[x] = P_1[x_2], P_2[x_2] = P_2[x_3], ...P_{i-1}[x_{i-1}] = P_{i-1}[x_i]\} \mid x \in X\}$

X is the set of all names among all input partitions: $X = \{x | x \in p, p \in \cup_i P_i\}$.

5.2.1.2 Transfer: Move

As mentioned earlier, movement is the operation we perform when we wish to kill old aliases while creating new ones.

Let T be the set of names that are being moved directly. This will contain the name on the left hand side of an assignment; the names of an argument to an unknown method or constructor call would also appear here, as they would be moved to the Dump. Let *target* be the name to which they will be moved.

To complete this transfer, we need to remove the moving names (and all names dependent on them–i.e., names which are fields of the names moving) from their partitions. This is complicated by our use of "implicit aliases"; the existence of a field $x.f$ in our representation implies that all names aliased with x have fields f aliased with $x.f$, but we do not record this explicitly. Moving x can thus inadvertently destroy the aliases of fields of aliases of x, unless we "save" the implicit aliases by making them explicit as we remove the names. We will do this by marking all of the names to be removed as we fill in implicit aliases. We then remove all marked names. Placeholder versions of the moving names are added to the target partition before this removal. (It may be worth noting that with a more explicit representation of objects, implicit aliases and the complication of saving them would not be necessary. This might be possible by storing field names as references on partitions, rather than on other names, as in [4].

The first helper function determines what partition a name currently resides in for a given partitioning of names. (We occasionally abuse the notation with $partition(\emptyset, E)$ as a verbose, type-incorrect way of saying "empty set".)

$$partition(x, E) = \begin{cases} P & \text{if } x \in P \text{ and } P \in E \\ \emptyset & \text{otherwise} \end{cases}$$

The next helper definition tells us if the second name is a series of field accesses with the target as the first name.

$$ancestor(x, y) = \begin{cases} true & \text{if } x = y \\ ancestor(x, y') & \text{if } y \neq x \text{ and } y = y'.f \\ false & \text{otherwise} \end{cases}$$

The next equations define the removal of names and the saving of implicit aliases. We iterate over all names that will be removed (T) and generate new names to save their implicit aliases if *canSave* suggests that this is required and possible. At the same time, we "mark" the names to be removed by adding them to a mark set. Auxiliary operation *gen* is defined below.

$(E, M, T) \rightarrow_{remove} (E', M', T')$ if

$$E' = E \setminus partition(t, E) \cup gen(t, E, M, T),$$
$$T' = (T \setminus \{t\}) \cup \{t.f | t.f \in \bigcup E\},$$
$$M' = M \cup \{t\}$$
$$T \neq \emptyset$$
where $t \in T$ is chosen arbitrarily.

The next definition tells us whether it is safe to save the implicit aliases created by name t on to name y. M is the set of "marked" names; these are names which are marked to be removed, and so it would not be useful to save implicit aliases onto them. The necessity of the other restrictions will be apparent during the proof of termination.

$$\begin{aligned} canSave(t, E, M, T, y) = & \exists f.(t = x.f) \\ & \text{and } y \in partition(x, E) \\ & \text{and } y \neq x \\ & \text{and } y \notin M \\ & \text{and } \forall t' \in T.\ \text{not } ancestor(t', y). \end{aligned}$$

$$gen(t, E, M, T) = \begin{cases} partition(t, E) \cup \{y.f\} & \text{if } \exists f.t = x.f \text{ and } y \text{ is such that} \\ & canSave(t, E, M, T, y) \\ partition(t, E) & \text{if } t \neq x.f \text{ or no such } y \text{ exists} \end{cases}$$

$\rightarrow^*_{remove}$ is the reflexive, transitive closure of $\rightarrow_{remove}$.

We use a second method of marking to ensure that the moving names are not removed from their new partition during this iteration. This is necessary because we must have the moving names in their new partition while we go through the process of removing and saving implicit aliases; handling a statement such as `x = x.a` illustrates why this is necessary. If `x.a` itself had fields with interesting aliases, those aliases need to be transferred to `x`, but this is only possible if `x` is aliased with `x.a` as we remove and save implicit aliases.

$T! = \{t + \text{“!”} | t \in T\}$

$x!$ and $t!$ below refer to any name containing an exclamation point.

$move(E, T, target) = \{P - M' - \{x!\} \cup \{t | t! \in P\} | P \in E'\}$
where E', M' are such that
$(E - partition(target, E) \cup \{partition(target, E) \cup T!\}, \emptyset, T) \rightarrow^*_{remove} (E', M', \emptyset)$

That some E', M' matching this description always exists will be the subject of section 5.3.1.

5.2.1.3 Example

To illustrate the way this operation works, we provide an example.

$E = \{\{a, b\}, \{a.f, c\}, \{d\}\}$

$E_next = move(E, \{a\}, d)$

Let E be the current environment. We wish to perform the $move(E, \{a\}, d)$ operation. $T = \{a\}$, $target = d$.

Initially, we add $a!$ to the partition of d. This does not have a necessary effect on the operation in this example, but if we were moving a to $a.f$, and $a.f.g$ had aliases, adding the placeholder name $a!$ would be crucial in giving us a place to save the aliases of $a.f$, despite the fact that no such location existed prior to the movement.

$(\{\{a, b\}, \{a.f, c\}, \{d, a!\}\}, \emptyset, \{a\})$

a is not a field name, so our condition $\exists f.t = x.f$ in $gen(a, E, \emptyset, \{a\})$ is false; therefore, $gen(a, E, \emptyset, \{a\} = partition(a, E)$. $E' = E \backslash partition(a, E) \cup partition(a, E) = E$. $T' = (T \setminus \{a\}) \cup \{a.f | a.f \in \bigcup E\} = \{a.f\}$, and $M' = \{a\}$.

$(\{\{a, b\}, \{a.f, c\}, \{d, a!\}\}, \emptyset, \{a\}) \rightarrow_{remove} (\{\{a, b\}, \{a.f, c\}, \{d, a!\}\}, \{a\}, \{a.f\})$

Now, $a.f$ is a field name. $canSave(a.f, E, \{a\}, \{a.f\}, b)$ is true; b is in the partition of a, $a.f$ is not an ancestor of b, b is not equal to a, $a.f$ is a field name, and b is not in M $(\{a\})$. Therefore, $gen(a.f, E, \{a\}, \{a.f\}) = partition(a, E) \cup \{b.f\}$. $E' = E \backslash partition(a, E) \cup (partition(a, E) \cup \{b.f\}$. $T' = (T \setminus \{a.f\}) \cup \{a.f.g | a.f.g \in \bigcup E\} = \emptyset$

(the variable field name f in the equation has been changed to g to distinguish it from the fixed field name f.

$(\{\{a,b\},\{a.f,c\},\{d,a!\}\},\{a\},\{a.f\}) \rightarrow_{remove} (\{\{a,b\},\{a.f,c,b.f\},\{d,a!\}\},\{a.f,a\},\emptyset)$

The T set is empty, so our $\rightarrow_{remove}$ operations are completed; after removing the names in the marked set and replacing the placeholder name, we have

$E_{next} = move(E,\{a\},d) = \{\{b\},\{c,b.f\},\{d,a\}\}$

5.2.1.4 Transfer: Merge

The merge operation merges all the target partitions, then fixes the partitions containing their fields by ensuring that for all x, y, and f,
$partition(x) = partition(y) \Rightarrow partition(x.f) = partition(y.f)$
if $x.f$ and $y.f$ exist.

The following operation ensures that this restriction holds by merging any two partitions where each contains a field access name of the same field and the targets of the two accesses are aliased. When this operation can no longer apply, the restriction must be satisfied.

$E \rightarrow_{merge}$ let $P_1 = partition(x.f, E)$, $P_2 = partition(y.f, E)$
in $(E - P_1 - P_2) \cup \{P_1 \cup P_2\}$
if $(P_1 \neq P_2$ & $partition(x, E) = partition(y, E))$ for some x and y.

Let $\rightarrow^*_{merge}$ be the reflexive, transitive closure of $\rightarrow_{merge}$.

$merge(E, S) = E'$ if
$(E - \{partition(t, E) | t \in S\} \cup (\bigcup_{t \in S} partition(t, E))) \rightarrow^*_{merge} E'$.
and there exists no E'' such that $E' \rightarrow_{merge} E''$.

5.2.2 Must-alias

The join function creates a partitioning where two names are in the same partition if and only if they were in the same partition in every incoming environment.

$$join(E_1, ...E_n) = \{ \ \{x\} \cup C | x \in \bigcup E_1 \text{ and}$$
$$C = \{y | \forall i. y \in partition(x, E_i)\}$$

The choice to define over $x \in E_1$ is arbitrary; since a name will only occur in the joined environment if it occurred in every incoming environment, any environment used must contain a superset of the interesting names in the joined environment.

5.2.2.1 Movement of names

Movement of names in the must-alias analysis uses the same definition as movement of names in the may-alias analysis.

5.2.2.2 Loss of control

The *inDump* helper function uses the may-alias analysis information to tell us whether a given object could have escaped at this point. This tells us what objects it is possible for unknown methods to modify.

$$inDump(x, E) = \begin{cases} true & \text{if } x \in partition(D*, E) \\ inDump(y, E) & \text{if } x = y.f \\ false & \text{otherwise} \end{cases}$$

The next two operations remove the aliases of all names that could be changed by an unknown method call: namely, field accesses on escaped objects.

$$kill_{loss}(E) = \{partition(x.f, E) | inDump(x, E) \text{ and } f \text{ is a field name } \}$$

$$lossOfControl(E) = E - kill_{loss}(E) \cup$$
$$\{P - \{x | inDump(x.f, E) \text{ and } x.f \in P\} | P \in kill_{loss}(E)\}$$

5.3 Proof of termination

5.3.1 $\rightarrow_{remove}$

The first thing we want to prove termination for is the "save implicit aliases" step, or the $\rightarrow_{remove}$ transfer function. We need to show that the $\rightarrow_{remove}$ operation will eventually terminate.

Observe that every application of $\rightarrow_{remove}$ will remove a name from the set T. Observe also that every name added to T will be longer (in terms of the number of field accesses) than the name removed from T. This means that the number of names of the shortest length in the T set will either decrease or stay constant at each step. (If the name chosen to examine during a step is one of the shortest, the number of names of that length will decrease; if a name which is not the shortest is chosen, that number will remain the same.)

Note also that we will never initiate a movement with both a field access and its parent in the T set. In the may-alias analysis, field names are never moved; in the must-alias analysis, field names can be moved, but there is no situation under which we will move multiple names.

This means that the T set can never contain a name which is an ancestor of another name in the set. This holds for the initial set because the initial set always contains only the name on the left hand side of an assignment. To show that it holds at each subsequent step, assume that T at the beginning of the step did not contain a name which was an ancestor of any other name in T. If T at the end of the step violates our condition, it must be the case that either one of the newly added names is an ancestor of an existing name, an existing name is an ancestor of a newly added name, or one newly added name is an ancestor of another. Note that all new names are field accesses on the name that was removed from T at this step (which we will refer to as t). If a new name is an ancestor of an existing name, then t must have been an ancestor of that existing name and the initial T violated our condition. If an existing name is an ancestor of a new name, it cannot be the direct parent of that name (because the direct parent is t

and t has been removed); therefore, the existing name must have been an ancestor of t and the initial T violated our condition. Two new names cannot be ancestors of each other by construction; all new names will be of the same length and end in different field accesses.

It follows that we will never add a name to the set T that has occurred in T before. Since every name is a field access of a name in the previous T, for any name in T at any point, the set T during all steps prior must have contained an ancestor of the name in question. If a newly added name was ever contained in a previous T, it must have coexisted with an ancestor of the new name (and the two cannot be the same, because the new name must have been added from the removal of a shorter name). We know that T can never contain a name and its ancestor at the same time, so it must be that names, once removed from T, never occur again.

By the construction of M, this is equivalent to saying that $T \cap M = \emptyset$.

Now, if we did not add new names to E during any step, this is sufficient to prove termination. There are a finite number of names in E, we remove at least one name from T at each step, and we never repeat names in T; therefore, after at most $|\bigcup_{p \in E}|$ steps, T must be empty.

However, we may add names to E during the iteration, and this means we can't assume the number of names in E remains finite.

We solve this by proving that any name added during the iteration can never enter the set T. For any given step, the only names that can be added after that step are names that have an ancestor in the current T set. The condition that we only add names to E when they do not have an ancestor in the current T thus prevents us from adding any names that will ever enter T.

Since we never add a name to E that will enter T, the set of names that can enter T must be fixed at the start of the iteration, and so must be finite. Since we remove one name at each step, never reexamine a name that we have removed, and have a finite number of names to examine, eventually our set of names to examine will be empty.

5.3.2 $\rightarrow_{merge}$

The termination proof here is simpler. At every application of $\rightarrow_{merge}$, the set of partitions has two partitions removed and one partition added. Since we start with a finite number of partitions, the number of times we can apply $\rightarrow_{merge}$ must be bounded by the number of partitions in the initial set.

5.4 Proof of correctness

5.4.1 May-alias

We need to show that our representation will never miss an alias that could hold at a program point.

The join function is simple to describe: the join of several incoming environments must contain all aliases that held in any incoming environment. The may-alias join function satisfies this condition: if two names a and b were aliased in any incoming environment, then they will appear in the same partition in the new environment under the condition $i = 2$, $x_1 = a$, $x_2 = b$, and $P_1 =$ the incoming environment in which a and b were aliased.

The remaining complexity of the join function is due to the fact that our environments must contain only one copy of each name in the program (or they would not be proper partitionings of the names). Thus, each partition must capture all aliases held by every name in that partition in any incoming environment. The join function captures this by extending the partition to cover all aliases of any name that is an alias of any name that has already been encountered (i.e., names that have been included in the partition already under a smaller i).

The transfer functions are more difficult, especially with the implied aliases. We will go over the statements possible in a flattened program and show why the operations of the analysis generate valid results for each statement.

We would also like to show that if an object can escape the current method, it will be aliased with the Dump (D*).

5.4.1.1 Initialization

If we allow the dump at the start of every method to contain the parameters of the method, *this*, and one instance of every field in the program, then we cover every alias that could exist at the start which could affect our analysis. Having everything in the dump amounts to a "We know nothing" state: the parameters could be aliased or not, the parameters could be *this*, and the fields of any of the parameters or *this* could be in an arbitrary state.

5.4.1.2 Movement

Before we prove the property on a per-statement basis, we should prove some lemmas about the move and merge operations.

For movement, we want to prove that after a move operation, the new environment contains all aliases present in the old environment except aliases of the names being moved, and also contains new aliases between the moving names and the target name.

To state this more clearly, all partitions which contained the names being moved, or field access names rooted at those names, must have had those names removed–but all other aliases, both implied and explicit, must remain. Additionally, to make things easier during our later proof of the must-alias analysis, we will also show that no new aliases are generated except those between the moving names and the target name (and fields thereof).

The construction of the mark set from the target set in the $\rightarrow_{remove}$ operation makes it clear that all of the moving names and their fields will be marked for removal. The target set initially contains the original moving names; every name that enters the target set will enter the mark set before the operation is complete (because the operation is not complete until the target set is empty, and the only time things are removed from the target set is when they enter the mark set). Each time a name is removed from the target set, the field names in the environment that are targeted on it are added to the target set. As seen in the proof of termination, we will never add a new field name to the

environment when one of its ancestors has been in the target set before; this prevents us from adding any field names to the environment which would match our criteria for names to be removed, and so we can see that every name rooted at one of the initial targets will be marked for removal.

At the same time, implied aliases must be preserved. An implied alias exists when two names may be aliased and a field name exists for one of the aliased names, but not the other. If we considered an infinite set of names in the environment, there would be no need for implied aliases as all names would be explicit; this gives us a good position from which to reason about the preservation of implied aliases. Effectively, if we took the old environment and expanded it to contain infinite names (and no implied aliases), and similarly expanded the environment after the $\rightarrow_{remove}$ operation, we want to show that the two expanded environments contain the same aliases (with the exception of those aliases involving the set of names to be moved).

$E \rightarrow_{expand} E'$
where $E' = (E - partition(x.f)) \cup \{partition(x.f) \cup \{y.f\}\}$ for some $x.f$ and y such that $y \in partition(x)$ and $y.f \notin \bigcup E$ and $x.f \in \bigcup E$.

The expansion of E is E after an infinite number of applications of this $\rightarrow_{expand}$ operation. This may or may not be bounded.

The remove operation preserves implicit aliases by instantiating some of the names that would be found in this infinite expansion. As it only instantiates names that will not be removed (do not have names in the initial target set as ancestors), all of the created names will be unmarked and will survive the removal process.

Aliases are only destroyed when names are removed. Names are only removed at the end of the move operation, and only if they are in the set of marked names. The effect of removing a name from the environment can be seen when we observe the infinite expansion of the environment. If we consider the names added by the expansion as separate from the "real" names in the environment, we can see that a change in the expanded environment only occurs when a "real" name is removed that is either a base

(non-field) name, or a field name where another name of the same field does not exist in the same partition with a target aliased to the removed name's target. Field names where such a name does exist are redundant, because the $\rightarrow_{expand}$ operation could replace them from the remaining names in the environment (x being the removed field's target and $y.f$ being the other field name).

The *canSave* condition and the *gen* set add names in order to make redundant the names being marked for removal. If a field name $x.f$ is being marked for removal and it is possible to locate a name y such that $y \in partition(x)$ (we'll examine the other conditions of *canSave* shortly), then we generate name $y.f$, making $x.f$ redundant and preserving the aliases that were implied by $x.f$ after its removal.

So, if all names being removed were field names, and they all met the conditions of *canSave*, there would be no aliases lost in the movement process at all. Clearly, however, this is not the case; we do wish to remove some aliases, and not all names meet those conditions. Let us examine in what situations a name will be marked for removal without a new name being added to save its implicit aliases.

First, the name could be a non-field name. Since the only names that are ever added to the set T (and, by extension, the set M) are field accesses on names already in T, the only non-field names that will enter M are the names in the initial T set. The aliases of these names are the aliases that we want to kill. Removing these names by themselves could cause implicit aliases to be lost; however, implicit aliases can only be caused by a base name if field names existed targeted on the base name. Since all field names targeted on the base names will be marked for removal before any names are removed, and the implicit aliases generated by those field names will be saved when they are marked, removing a non-field name does not destroy any aliases other than those that should be killed by the movement.

Second, the name could be a field name, but there could be no y that meets the conditions of *canSave*. The *canSave* conditions are intended to ensure that there is a suitable name to save the implicit aliases on. A name used to save implicit aliases must first be in the same partition as the target of the field being removed; the purpose of this

restriction is obvious. Saving implicit aliases onto a target in a different partition would create spurious new aliases without saving the old ones, and if there are no names in the partition of the target, there must be no implicit aliases generated by the field name being removed. (Potentially, if the target is also a field name, it is possible that there could be other implicit aliases with the target with other fields whose aliases would be destroyed by removing this field; however, as we are guaranteed to mark the target of this field before marking this field, at least one implicit alias of that target would have been instantiated at that time, and so the target can only be alone in a partition if there are no implicit aliases with it.) The conditions of $y \notin M$ and $\forall t' \in T$, not $ancestor(t', y)$ serve to ensure that y was not marked for removal and will not be marked for removal during a later stage of the marking process. If the only names aliased with the target are names marked for removal, then it is safe to destroy their implicit aliases because they were in the tree of names whose aliases we want to remove.

We have seen, then, that the only aliases removed by the movement are those that involve the names being removed. We can also see from the infinite expansion that no spurious new aliases are generated; the only additions to partitions during the movement are the additions of the moving names to the target partition, and the addition of names used for saving implicit aliases. Implicit alias-saving names only illuminate hidden portions of the infinite expansion, and therefore do not add any new aliases; the moving names themselves only generate new aliases reflecting the effect of the movement (namely, that the moving names are now aliased with the name they were being moved to).

5.4.1.3 Merge

We need to prove that a merge operation preserves all aliases in the old environment, and adds new aliases for the names being merged. (Spurious aliases are to be accepted; the merge operation is not used in the must-alias analysis, so it will not matter for correctness.)

The $\rightarrow_{merge}$ operation removes two partitions from the environment and replaces them with their union. It is clear that the set of names in the environment does not change, and that if two names were in the same partition in the environment before the operation, they will be in the same partition in the environment after the operation. Therefore, applications of $\rightarrow_{merge}$ do not destroy any aliases.

The initialization of the merge operation also involves replacing a number of partitions with their union, so by the same argument, it does not destroy any aliases, either. The union in the initialization ensures that the target names will all be in the same partition in the resulting environment; therefore, at the end of a merge, aliases will exist between the names being merged.

5.4.1.4 x = y

The transfer here needs to retain all aliases in the incoming environment (except, perhaps, those involving x), and add a new alias between x and y.

If x was a simple name, it is safe to kill the existing aliases of x. Our movement operation, by the proof above, will result in killing the aliases of x, creating a new alias between x and y, and preserving all other aliases.

If x was a field access, the merge operation will create a new alias between x and y and destroy no other aliases; the same holds if x was an array access.

5.4.1.5 x = (class)y

A cast operation has no effect on the aliasing of names, so our handling of this case as `x = y` is valid if our handling of `x = y` is valid.

5.4.1.6 x = new c(...)

There will be no existing name for the new object, and so the escape of the parameters is the only change that must be recorded for validity.

If the constructor call was inlined, then there is no escape (any escape of parameters that occurs in the constructor will be visible in the inlined constructor body). If x is a

simple name, then it is safe to kill the prior aliases of x, and the movement will accomplish this without destroying other aliases; otherwise, we do nothing, and so cannot destroy any necessary aliases.

If the constructor call was not inlined, then the call represents an escape of the parameters. The newly created object could also potentially be aliased with any field of an escaped name, so the assignee has escaped as well. To represent this escape, we alias the escaping names with any other potentially escaped names by moving or merging them to the Dump. The merge operation destroys no aliases, and so must be valid; the movement operation only occurs if x was a simple name, in which case it is safe to destroy the existing aliases of x.

5.4.1.7 x = method(...)

This uses the same operations as the non-inlined constructor call, and the logic is the same.

5.4.1.8 Inner class declaration

The statements of the inner class will be visited separately, but that traversal will only be to determine alias information for the bodies of the methods in the inner class. The effect on the outer level is identical to a method call–we assume that we do not know what may happen inside the body of the class. Any object that is captured by the inner class could be manipulated whenever we lose control, and so we consider them escaped. Merging every outer name used in the inner class into the Dump preserves all existing aliases and correctly creates the new aliases indicating that the names have escaped.

5.4.2 Must-alias

Correctness in the must-alias analysis is dependent upon not generating any spurious aliases. Therefore, we will show that every alias generated by the system is valid.

5.4.2.1 Initialization

Initialization is clearly valid, as the initial state contains no aliases.

5.4.2.2 Join

The join function creates an environment in which two names are in the same partition if and only if they were in the same partition in every incoming environment. This ensures that the only aliases in the joined environment are those that will hold on every incoming path.

5.4.2.3 Loss of control

For our final intermediate operation, we will show that the *lossOfControl* feature removes all aliases that could be invalidated by the execution of unknown code. We do assume that the program is not multithreaded, and we assume that we have available the information from the may-alias analysis for the program.

Unknown code can only alter the fields of objects which have escaped the current method. (Array contents could also be changed, but we do not consider array contents in the must-alias analysis.) The may-alias analysis provides us with a set of the objects at a particular point which could have escaped the current method: all objects that may be aliased with the Dump may have escaped. It can be seen from the definition of *inDump*, then, that *inDump* will be true for all names which could have escaped the current function.

The effect of the final *lossOfControl* equation is to remove from the environment all field names for which *inDump* is true of the target of the field. This removes all aliases of names which could be modified by external code.

5.4.2.4 x = y

When we encounter this statement, we use the Movement operation to create a new alias between x and y. We know that this alias must hold immediately after this statement;

our earlier examination of the movement option shows that the movement will not create any spontaneous invalid aliases.

5.4.2.5 x = (class)y

Again, we ignore casting, and so the previous proof suffices.

5.4.2.6 x = new c($y_1, y_2, ...y_n$)

Ignoring constant-names for the moment, the effect of this statement on x is to invalidate all prior alias information on x. The movement we perform here will remove x from its prior partition, killing those aliases. (Our *constant* operator will never give us a constant-name already in the environment, so this movement will never create new aliases except between x and the constant-name.)

Additionally, if the constructor has not been inlined, then it represents execution of code not in our perception. This potentially endangers alias information about some objects; however, only the mutable fields of objects which have escaped will be accessible to the external code. Therefore, transforming the environment through an application of the *lossOfControl* operation will remove the potentially invalidated aliases.

5.4.2.7 x = c

As with $x = y$, the only effect on our alias information from this statement is that it invalidates the aliases of x; the movement to the fresh constant-name will suffice to destroy them.

5.4.2.8 x = method($y_1, y_2, ...y_n$)

By the same reasoning as the non-inlined constructor call, handling a loss of control will kill the aliases endangered by the execution of the method; moving x to an empty partition will kill the aliases of x.

5.4.2.9 Inner class declaration

The escape of variables bound in the inner class has already been handled by the may-alias analysis; the only effect on the must-alias analysis of those escapes will be seen through the may-alias Dump partition. The only remaining effect on the must-alias analysis of an inner class declaration is that of the constructor call (if the inner class was an anonymous class creation); this has the same effect as a normal method or constructor call, and the same operation suffices to handle it.

Chapter 6

Examples

6.1 Examples

6.1.1 Rewriter application: Iterators

This example will illustrate how temporary constructor inlining, alias analysis, and the other rewriters are used to resolve one of our common cases: iteration over a known list.

Handling iterators was one of the major bottlenecks that we encountered while using the early rewriting system. Jumbo uses a number of custom collection classes that support iteration. This aids in rewriting because it gives us access to (and lets us control) the source code for the collections. These collection classes are used in many places during compilation: members of a class, classes in a file, statements in a block, parameters or arguments of a method, and interfaces of a class are all stored in objects of the container classes. After our restructuring of the compiler, the compilation environment itself became one of these collections. Almost all of the accesses to these collections are handled through iterator loops.

Iteration over these collections was a major (initially) irreducible part of the generators we were attempting to rewrite, and our inability to resolve them often blocked further rewriting progress by denying us information that could have been used for subsequent rewrites (and which should have been theoretically available, being based on static

information). Iterators by their nature must have non-final fields, and must change the value of those fields during iteration. Often, the construction of the entire list was visible (i.e., the initial list creation and all additions to the list were in the current method and dominated the use), but the iteration over the list could not be resolved–even if the list was empty. We had the ability to unroll the iteration loop enough times to cover all members of the list, but the non-final fields of the iterator and the assignments to the iterator field prevented us from resolving any of the accesses to the list elements. Additionally, resolving a list element required at least two levels of field accesses–once through the field of the iterator to reach the container, and another through the field of the container that holds the contents; neither Untupling nor FieldValue (the only two preexisting rewriters that dealt with fields) could handle two-level field accesses.

Resolving these situations was one of the major goals of the alias analysis and temporary constructor inlining. Together, they can successfully resolve most iterator situations, if the list construction is in view. We go over a simple example here.

6.1.1.1 Original file

This is the complete example file we will be using, before rewriting.

```
import uiuc.Jumbo.Util.*;
import java.util.Iterator;

class Parameter
{
    public final String name;
    public Parameter(String s)
    {
         name = s;
    }

    public Parameter normalize()
    {
        return new Parameter(name + "-normalized!");
    }
}
```

```
public class Test1
{
    public MonoList getNormalizedParameters()
    {
        final MonoList l1 =
            new EmptyLinkedMonoList()
            .add(new Parameter("p1"))
            .add(new Parameter("p2"))
            .add(new Parameter("p3"));
        MonoList rval = new EmptyLinkedMonoList();
        for(Iterator i = l1.iterator(); i.hasNext();)
        {
            Parameter p = (Parameter)i.next();
            rval = rval.add(p.normalize());
        }
        return rval;
    }
}
```

We will be rewriting to optimize the `getNormalizedParameters` function. The Parameter class is simplified and the code is taken out of context, but the operation is very similar to operations which can be found in several places in the Jumbo compiler.

6.1.1.2 Flattened code

After flattening, the `getNormalizedParameters` method looks like this.

```
  public uiuc.Jumbo.Util.MonoList getNormalizedParameters()
  {
    final uiuc.Jumbo.Util.MonoList l1;
    final uiuc.Jumbo.Util.EmptyLinkedMonoList flatten_3;
    flatten_3 = new uiuc.Jumbo.Util.EmptyLinkedMonoList();
    final Parameter flatten_4;
    flatten_4 = new Parameter("p1");
    final uiuc.Jumbo.Util.MonoList flatten_5;
    flatten_5 = flatten_3.add(flatten_4);
    final Parameter flatten_6;
    flatten_6 = new Parameter("p2");
    final uiuc.Jumbo.Util.MonoList flatten_7;
    flatten_7 = flatten_5.add(flatten_6);
    final Parameter flatten_8;
    flatten_8 = new Parameter("p3");
```

```
  l1 = flatten_7.add(flatten_8);
  uiuc.Jumbo.Util.MonoList rval;
  rval = new uiuc.Jumbo.Util.EmptyLinkedMonoList();
  java.util.Iterator i;
  i = l1.iterator();
  while (true)
  {
    final boolean flatten_11;
    flatten_11 = i.hasNext();
    if (flatten_11)
    {
      Parameter p;
      final java.lang.Object flatten_9;
      flatten_9 = i.next();
      p = ((Parameter)flatten_9);
      final Parameter flatten_10;
      flatten_10 = p.normalize();
      rval = rval.add(flatten_10);
    } else break;
  }
  return rval;
}
```

(The `normalize` and constructor methods of `Parameter` have also been flattened, but their changes are not relevant to the example at the moment.)

6.1.1.3 After inlining

Next, we inline the calls to `add`, and use the automatic code-reducing rewriters to clean up the code.

```
public uiuc.Jumbo.Util.MonoList getNormalizedParameters()
{
  final Parameter flatten_4;
  flatten_4 = new Parameter("p1");
  final Parameter flatten_6;
  flatten_6 = new Parameter("p2");
  final Parameter flatten_8;
  flatten_8 = new Parameter("p3");
  final uiuc.Jumbo.Util.LinkedMonoList gen159_flatten_50;
  gen159_flatten_50 = new uiuc.Jumbo.Util.LinkedMonoList(flatten_8);
  final uiuc.Jumbo.Util.LinkedMonoList gen159_flatten_51;
```

```
  gen159_flatten_51 =
      new uiuc.Jumbo.Util.LinkedMonoList(flatten_6, gen159_flatten_50);
  final uiuc.Jumbo.Util.LinkedMonoList gen158_flatten_48;
  gen158_flatten_48 =
      new uiuc.Jumbo.Util.LinkedMonoList(flatten_4, gen159_flatten_51);
  uiuc.Jumbo.Util.MonoList rval;
  rval = new uiuc.Jumbo.Util.EmptyLinkedMonoList();
  java.util.Iterator i;
  i = gen158_flatten_48.iterator();
  while (true)
  {
    final boolean flatten_11;
    flatten_11 = i.hasNext();
    if (flatten_11)
    {
      Parameter p;
      final java.lang.Object flatten_9;
      flatten_9 = i.next();
      p = ((Parameter)flatten_9);
      final Parameter flatten_10;
      flatten_10 = p.normalize();
      rval = rval.add(flatten_10);
    } else break;
  }
  return rval;
}
```

The list construction is now fully in view, in that we can see constructor calls for every link in the list.

6.1.1.4 Unrolling

We have next inlined the `hasNext` and `next` calls inside the loop, and then unrolled the loop once. (Inlining the calls after unrolling the loop would look cleaner, but would require using the Inlining rewriter after every unrolled iteration; inlining the calls before unrolling allows the inlining to be performed only once.)

```
public uiuc.Jumbo.Util.MonoList getNormalizedParameters()
{
  final Parameter flatten_4;
```

```
flatten_4 = new Parameter("p1");
final Parameter flatten_6;
flatten_6 = new Parameter("p2");
final Parameter flatten_8;
flatten_8 = new Parameter("p3");
final uiuc.Jumbo.Util.LinkedMonoList gen159_flatten_50;
gen159_flatten_50 = new uiuc.Jumbo.Util.LinkedMonoList(flatten_8);
final uiuc.Jumbo.Util.LinkedMonoList gen159_flatten_51;
gen159_flatten_51 =
    new uiuc.Jumbo.Util.LinkedMonoList(flatten_6, gen159_flatten_50);
final uiuc.Jumbo.Util.LinkedMonoList gen158_flatten_48;
gen158_flatten_48 =
    new uiuc.Jumbo.Util.LinkedMonoList(flatten_4, gen159_flatten_51);
uiuc.Jumbo.Util.MonoList rval;
rval = new uiuc.Jumbo.Util.EmptyLinkedMonoList();
final uiuc.Jumbo.Util.LinkedMonoList
            .LinkedMonoIterator gen160_flatten_54;
gen160_flatten_54 =
    new uiuc.Jumbo.Util.LinkedMonoList
                    .LinkedMonoIterator(gen158_flatten_48);
unroll1_label:
{
  final uiuc.Jumbo.Util.LinkedMonoList unroll1_gen162_flatten_63;
  unroll1_gen162_flatten_63 = gen160_flatten_54.a;
  final boolean unroll1_gen162_flatten_64;
  unroll1_gen162_flatten_64 = (unroll1_gen162_flatten_63!=null);
  if (unroll1_gen162_flatten_64)
  {
    Parameter unroll1_p;
    final uiuc.Jumbo.Util.LinkedMonoList unroll1_gen161_flatten_66;
    unroll1_gen161_flatten_66 = gen160_flatten_54.a;
    final boolean unroll1_gen161_flatten_67;
    unroll1_gen161_flatten_67 = (unroll1_gen161_flatten_66==null);
    if (unroll1_gen161_flatten_67)
    {
      final java.util.NoSuchElementException
                    unroll1_gen161_flatten_65;
      unroll1_gen161_flatten_65 =
                    new java.util.NoSuchElementException();
      throw unroll1_gen161_flatten_65;
    }
    final java.lang.Object unroll1_gen161_d;
    final uiuc.Jumbo.Util.LinkedMonoList unroll1_gen161_flatten_68;
```

```
    unroll1_gen161_flatten_68 = gen160_flatten_54.a;
    unroll1_gen161_d = unroll1_gen161_flatten_68.data;
    final uiuc.Jumbo.Util.LinkedMonoList unroll1_gen161_flatten_69;
    unroll1_gen161_flatten_69 = gen160_flatten_54.a;
    gen160_flatten_54.a = unroll1_gen161_flatten_69.next;
    unroll1_p = ((Parameter)unroll1_gen161_d);
    final Parameter unroll1_flatten_10;
    unroll1_flatten_10 = unroll1_p.normalize();
    rval = rval.add(unroll1_flatten_10);
  } else break unroll1_label;
  while (true)
  {
    final uiuc.Jumbo.Util.LinkedMonoList gen162_flatten_63;
    gen162_flatten_63 = gen160_flatten_54.a;
    final boolean gen162_flatten_64;
    gen162_flatten_64 = (gen162_flatten_63!=null);
    if (gen162_flatten_64)
    {
      Parameter p;
      final uiuc.Jumbo.Util.LinkedMonoList gen161_flatten_66;
      gen161_flatten_66 = gen160_flatten_54.a;
      final boolean gen161_flatten_67;
      gen161_flatten_67 = (gen161_flatten_66==null);
      if (gen161_flatten_67)
      {
        final java.util.NoSuchElementException gen161_flatten_65;
        gen161_flatten_65 = new java.util.NoSuchElementException();
        throw gen161_flatten_65;
      }
      final java.lang.Object gen161_d;
      final uiuc.Jumbo.Util.LinkedMonoList gen161_flatten_68;
      gen161_flatten_68 = gen160_flatten_54.a;
      gen161_d = gen161_flatten_68.data;
      final uiuc.Jumbo.Util.LinkedMonoList gen161_flatten_69;
      gen161_flatten_69 = gen160_flatten_54.a;
      gen160_flatten_54.a = gen161_flatten_69.next;
      p = ((Parameter)gen161_d);
      final Parameter flatten_10;
      flatten_10 = p.normalize();
      rval = rval.add(flatten_10);
    } else break;
  }
}
```

```
    return rval;
  }
```

The unrolled iteration is visible in the block labeled `unroll1_label`.

All of the steps up to this point have been handled by existing rewriters. Now, however, we have reached a point where the previous rewriters are incapable of proceeding. The information to resolve the loop is available, but requires tracing through the constructors of the iterator and the lists, and relies on information coming from a non-final field (the `iterator.a` field). We now use temporary constructor inlining and alias analysis to get past this hurdle.

6.1.1.5 Temporary constructor inlining

Our first new step is to temporarily inline the constructors for the iterator and the first link in the list (note that our linked lists are constructed in reverse order).

```
gen163_gen158_flatten_48 =
    new uiuc.Jumbo.Util.LinkedMonoList(flatten_4, gen159_flatten_51)
{
    java.lang.Object.<init>();
    gen163_gen158_flatten_48.data = null;
    gen163_gen158_flatten_48.next = null;
    gen163_gen158_flatten_48.data = flatten_4;
    gen163_gen158_flatten_48.next = gen159_flatten_51;
};

gen0_gen160_flatten_54 =
    new uiuc.Jumbo.Util.LinkedMonoList
                .LinkedMonoIterator(gen163_gen158_flatten_48)
{
    java.lang.Object.<init>();
    gen0_gen160_flatten_54.a = null;
    gen0_gen160_flatten_54.a = gen163_gen158_flatten_48;
};
```

Both of the bracketed sections here represent the temporarily inlined constructor bodies.

The extra field initializations are present to ensure that fields always receive at least the default initialization; even if the field will be initialized explicitly in the constructor, virtual method calls from superconstructors can access the field before its initialization, so the default initialization is still necessary. (The UnusedFieldAssign rewriter is unable to remove these assignments because we do not have use-def information for fields and cannot tell that the field assignments are overridden before being used; it would be possible to create a rewriter for this situation, but since temporarily inlined constructors will always be removed before compilation, and the alias analysis will correctly handle the overriding field assignments by only using the information from the latest assignment to these fields, such a rewriter would neither optimize this code nor enable other rewrites.)

The `java.lang.Object.<init>` "method call" is a placeholder for the superconstructor call to `java.lang.Object`. These superconstructor call placeholders are generated whenever we do not have access to the source code for a superclass; the `java.lang.Object` placeholder receives special treatment by the must-alias analysis in that it is assumed not to cause the escape of the object being created, and to alter no fields of any objects (in the terminology of our alias analysis, we do not need to consider a call to `java.lang.Object.<init>` as a *lossOfControl*). This assumption gives us the ability to eliminate losses of control from object creations, increasing the ability of our alias analysis to propagate information between assignments and uses of non-final fields on escaped objects; it may be useful and safe to extend this assumption to other Java classes that are commonly used as bases, but `Object` has been sufficient for our purposes.

6.1.1.6 Must-alias analysis

We then run our must-alias analysis on the code. The may-alias analysis is run first, but the only pertinent result from it is acknowledgement that the iterator does not escape; this allows the must-alias analysis to track the information in the non-final `.a` field.

At the beginning of the unrolled loop iteration block, the must-alias environment looks like this:

```
[ LinkedMonoIterator(gen163_gen158_flatten_48)#10  gen0_gen160_flatten_54]
```

```
[ EmptyLinkedMonoList()#9  rval]
[ Parameter("p2")#2  flatten_6]
[ gen159_flatten_50  LinkedMonoList(flatten_8)#4]
[ Parameter("p3")#3  flatten_8]
[ Parameter("p1")#1  flatten_4  gen163_gen158_flatten_48.data]
[ gen163_gen158_flatten_48.next  gen159_flatten_51
    LinkedMonoList(flatten_6, gen159_flatten_50)#5]
[ gen163_gen158_flatten_48  gen0_gen160_flatten_54.a
    LinkedMonoList(flatten_4, gen159_flatten_51)#6]
```

The entries marked with numbers (`#N`) are constant names representing objects created by the program, as mentioned in section 5.1.2.3. These are the important entries for the resolution of the null comparisons; both of the checks for null in the unrolled section are for `gen0_gen160_flatten_54.a` (or variables which will be aliased with `gen0_gen160_flatten_54.a` at the time of the check), and so we can see that the AliasObjectEquality rewriter will resolve those comparisons successfully. We can also see that `gen163_gen158_flatten_48.data` is aliased with `Parameter("p1")`; we can now follow the assignment through `gen161_d` to `p`, which will allow us to inline the call to `normalize`.

6.1.1.7 After using alias information

AliasObjectEquality replaces the two null checks; AliasFieldReplacement replaces the instances of `gen0_gen160_flatten_54.a` and the `.data` field access on it. After running the cleanup rewriters, the loop looks like this:

```
final uiuc.Jumbo.Util.LinkedMonoList.LinkedMonoIterator
            gen0_gen160_flatten_54;
gen0_gen160_flatten_54 =
    new uiuc.Jumbo.Util.LinkedMonoList
            .LinkedMonoIterator(gen163_gen158_flatten_48);
gen0_gen160_flatten_54.a = gen159_flatten_51;
final Parameter unroll1_flatten_10;
unroll1_flatten_10 = flatten_4.normalize();
rval = rval.add(unroll1_flatten_10);
while (true)
{
```

```
    final uiuc.Jumbo.Util.LinkedMonoList gen162_flatten_63;
    gen162_flatten_63 = gen0_gen160_flatten_54.a;
    final boolean gen162_flatten_64;
    gen162_flatten_64 = (gen162_flatten_63!=null);
    if (gen162_flatten_64)
    {
      Parameter p;
      final boolean gen161_flatten_67;
      gen161_flatten_67 = (gen162_flatten_63==null);
      if (gen161_flatten_67)
      {
        final java.util.NoSuchElementException gen161_flatten_65;
        gen161_flatten_65 = new java.util.NoSuchElementException();
        throw gen161_flatten_65;

      }
      final java.lang.Object gen161_d;
      gen161_d = gen162_flatten_63.data;
      gen0_gen160_flatten_54.a = gen162_flatten_63.next;
      p = ((Parameter)gen161_d);
      final Parameter flatten_10;
      flatten_10 = p.normalize();
      rval = rval.add(flatten_10);
    } else break;
  }
```

The unrolled iteration is reduced to only a few lines.

After unrolling the loop further and repeating the process for the other two elements of the list, the method is much cleaner.

```
  public uiuc.Jumbo.Util.MonoList getNormalizedParameters()
  {
    final Parameter flatten_4;
    flatten_4 = new Parameter("p1");
    final Parameter flatten_6;
    flatten_6 = new Parameter("p2");
    final Parameter flatten_8;
    flatten_8 = new Parameter("p3");
    uiuc.Jumbo.Util.MonoList rval;
    rval = new uiuc.Jumbo.Util.EmptyLinkedMonoList();
    final Parameter unroll1_flatten_10;
    unroll1_flatten_10 = flatten_4.normalize();
```

```
    rval = rval.add(unroll1_flatten_10);
    final Parameter unroll2_flatten_10;
    unroll2_flatten_10 = flatten_6.normalize();
    rval = rval.add(unroll2_flatten_10);
    final Parameter unroll3_flatten_10;
    unroll3_flatten_10 = flatten_8.normalize();
    rval = rval.add(unroll3_flatten_10);
    return rval;
  }
```

We can then use the other rewriters to inline and clean up the calls to `normalize` and `add`.

Chapter 7

Conclusion

7.1 Conclusions

Our alias analysis and temporary constructor inlining were successful at resolving many of the problems we encountered during rewriting with the propagation of values through fields. Alias information increased the effectiveness of many rewriters, as well as enabling the creation of some new rewriters, and it allowed us to progress through several common situations which had blocked further rewriting in the past.

However, some of our design decisions reduced our effectiveness, in particular the choice to develop a strictly intraprocedural algorithm. We felt that we would not always have access to the source code for all possible called procedures, and decided that all interprocedural aspects would be handled transparently to the alias analysis system by inlining; since the inlining process is not automated, however, the success of the alias analysis is often dependent on the amount of time spent by the user on inlining. An interprocedural analysis may have been able to reduce this issue, and would likely increase the effectiveness of the algorithm in situations where we were willing to assume that we had access to the code of all possible methods that could be called at a site.

Bibliography

[1] Lars Clausen. *Optimizations in Distributed Run-Time Compilation.* PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[2] Sam Kamin and T. Baris Aktemur. Mumbo: A rule based implementation of a run-time program generation language. *6th International Workshop on Rule-Based Programming*, pages 27–50, April 2005.

[3] Samuel Kamin, T. Baris Aktemur, and Philip Morton. Source-level optimization of run-time program generators. In *Springer Lecture Notes in Computer Science*, volume 3676, pages 293–308, Sept 2005.

[4] Jongwook Woo, Jean-Luc Gaudiot, and Andrew L. Wendelborn. Alias analysis in java with reference-set representation for high-performance computing. *Int. J. Parallel Program.*, 32(1):39–76, 2004.