Tree Form: An Intermediate Representation for Retargetable Optimizing Compilers

by

Duncan G. Bryce

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

~~January 25, 2000~~
February 2000

Author____
Department of Electrical Engineering and Computer Science
January 25, 2000

Certified by____________________________________
Martin Rinard
sor

Accepted by____________________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Tree Form: An Intermediate Representation for
Retargetable Optimizing Compilers
by
Duncan G. Bryce

Submitted to the
Department of Electrical Engineering and Computer Science

January 25, 2000

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer [Electrical] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Tree form is a low-level, architecture-independent intermediate representation (IR) used by the Flex java-to-hardware compiler. Using a tree-based IR allows for the automatic construction of high-quality code generators for a substantial set of computer architectures, from only the precise specification of these architectures. This paper provides a thorough description of Tree form: its construction, programmatic manipulation, optimization, and utility.

Thesis Supervisor: Martin Rinard
Title: Assistant Professor, MIT Laboratory for Computer Science

Acknowledgements

I'd first like to thank Martin Rinard, for providing me with an incredible learning experience. Second, I must thank C. Scott Ananian, for having an answer to every question, no matter how obscure. Just to make sure I don't forget anybody, I'd like to extend a blanket "thank you" to all of the members of the Flex Compiler Group. Each one of you guys is brilliant.

I'd like to thank Sandia Ren, Laurie Qian, and Boo, for giving me food, and Sidney, for her constant support. Lastly, I'd like to thank my family: Andrew, Peter, Mom, and Dad. Without them I would never have made it this far.

1 Introduction

The Flex compiler infrastructure is a retargetable Java-to-hardware optimizing compiler, implemented in Java. To be highly optimizing, Flex requires an intermediate representation (IR) that is conducive to the efficient application of code optimization algorithms. To be highly retargetable, Flex requires an IR that can be conveniently translated into real machine language, for as many architectures as possible[1].

Unfortunately, IRs conducive to many important forms of code optimization are not generally low-level enough to be conveniently translated into machine code. Thus, to remedy this situation, the Flex compiler uses multiple IRs:

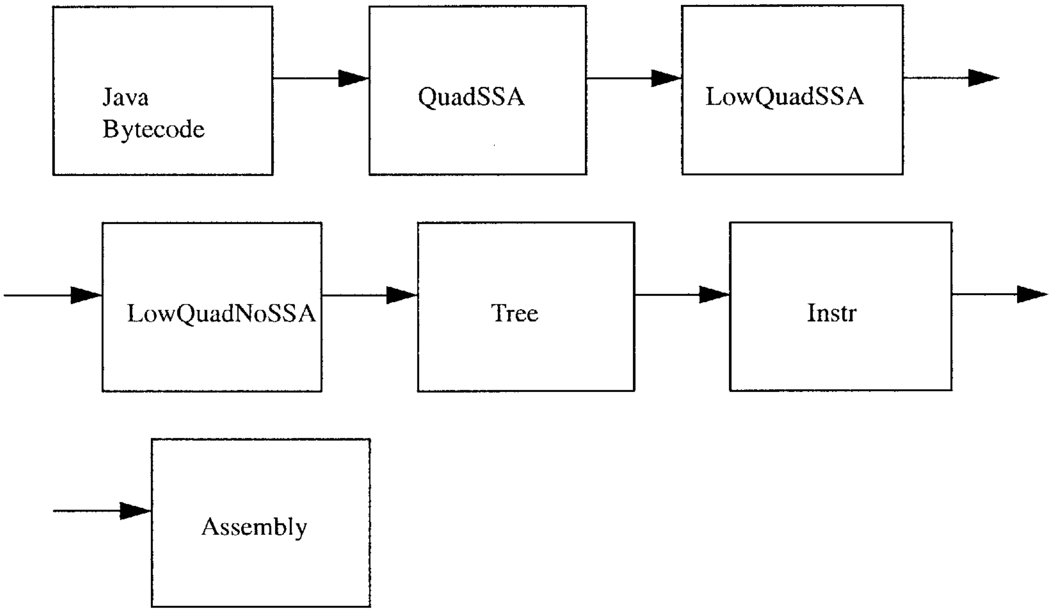


Figure 1: Phases of the Flex Compiler

The goal of this paper twofold. First, it is an attempt to provide a thorough introduction to *Tree* form, Flex's tree-based IR. Second, it is a description of some of the more useful work I have done on Tree form over the last couple of years. While neither goal is exhaustively explored, I've given precedence to the first: many aspects of tree form which are discussed herein were not implemented by me, and I omit discussion of software I've written which does not further the reader's understanding of tree form.

Section 2 provides the background necessary to understand Tree form. Section 3 provides the motivation for using a tree-based representation. Section 4 provides a specification of Tree form. Section 5 provides an introduction to manipulating Tree form programmatically. Section 6 discusses how Tree form can be used to represent static class data. Section 7 examines the process of constructing Tree form from LowQuadSSA form. Section 8 describes the process of analyzing and optimizing the Tree form. Finally, sections 9 and 10 discuss future directions and provide a conclusion.

2 Background

This section contains the background necessary to understand and effectively use Tree form.

First, a very brief introduction to the layout of the Flex Compiler Infrastructure is provided, as a general familiarity of the available features of the Flex infrastructure is necessary to be able to use any of its IRs. Following this introduction is a brief description of the other IRs used in the Flex compiler. This will aid in the understanding of the construction of Tree form from higher-level IRs, and will allow for the observation of key points where Tree form differs from Flex's other representations. Lastly, the *Visitor* design pattern is introduced, as the vast majority of transformations in the Flex compiler use it as their foundation.

2.1 Layout of the Flex Compiler Infrastructure

The Flex Compiler Infrastructure is implemented almost entirely in Java (the exceptions to this rule being the native libraries, and runtime system). Classes in the infrastructure are packaged into the Flex Compiler class hierarchy at some location based on their function. For instance, the classes pertaining to intermediate code representation are all in a sub-package of `harpoon.IR`[1], classes pertinent to code analysis reside in a sub-package of `harpoon.Analysis`, etc.

[1] The original name of the Flex Compiler Infrastructure was Harpoon.

The important high-level packages are:

Package	Function
`harpoon.Analysis`	contains analysis routines for a variety of IRs
`harpoon.Backend`	contains the analysis and code generation routines necessary to generate platform-specific machine code
`harpoon.ClassFile`	contains representations of class hierarchy, object structure, and method code
`harpoon.Interpret`	contains interpreters for various IRs
`harpoon.IR`	contains implementations of various IRs
`harpoon.Main`	contains the command line interfaces for running the compiler and its various utility programs
`harpoon.Temp`	contains the classes for uniquely generating and manipulating temporaries that represent variables or symbolic addresses
`harpoon.Util`	contains implementations of general-purpose algorithms and types

Figure 2: Important Packages in the Flex Hierarchy

2.2 Intermediate Representations of the Flex Compiler

2.2.1 QuadSSA

QuadSSA is a quadruple-based representation in static single-assignment (SSA) form.

By quadruple-based, we mean that the elements of the IR are of the form $a \leftarrow b \oplus c$ [1]. The source operands b and c are combined by an arbitrary binary operator, and written to the destination operand, a. There are two kinds of source operands:

constant operands represent constant data, such as numeric literals or labels, and are specified through Java Objects or primitives. *Variable* operands are of type `harpoon.Temp.Temp`. A `Temp` is a "virtual register". `Temps` are used in all of the Flex IRs to represent variable storage.

SSA form "is a relatively new intermediate representation that effectively separates the values operated on in a program from the locations they are stored in, making possible more effective versions of several optimizations" [2]. A representation which uses SSA form ensures that every variable assigned a value is the target of only one assignment. In the absence of branches or join points, it is easy to translate to SSA form through a subscripting process found in most modern compiler texts [1, 2]. The introduction of these elements requires the use of ϕ-functions at the join points, which "magically" select the correct one of multiple assignments to the same variable. To clarify, consider the following pseudo-code:

```
i = read();
if (i > 0) {
        i = 1;
} else {
        i = 2;
}
print(i);
```

This would translate to the following SSA form:

```
i0 = read();
if (i0 > 0) {
        i1 = 1;
} else {
        i2 = 2;
}
i3 = φ(i1, i2);
print(i3);
```

In the assignment to `i3`, the ϕ-function magically selects the correct source operand, and writes it to the destination operand. Crucial to the understanding of the ϕ–function is the criterion by which the ϕ–function magically selects the correct source operand. Each ϕ–

function can have any number of statements as its direct predecessor in the control-flow graph. The number of statements which directly precede the ϕ–function is called its *arity.* The number of source operands a ϕ–function must select from is equal to its arity. Each source operand is live only at *one* of the direct predecessors of the ϕ–function. The ϕ–function will magically select the operand that was live when the ϕ–function was reached (i.e., will select based on the execution path that was actually taken).

The complement of the ϕ-function is the σ-function. These operators are used at branch points. Rather than "magically" selecting a value from some set of possible values as the ϕ–functions do, the σ–function maps one value to a set of values, one value for each branch destination:

```
int x = 0;
switch (read()) {
     case 0:   x++; break;
     case 1:   x--; break;
     default:  x *= 3;
}
print(x);
```

Would translate to:

```
int x = 0;
σ(x1, x2, x3) = x;
switch (read()) {
     case 0:   x4 = x1 + 1; break;
     case 1:   x5 = x2 - 1; break;
     default:  x6 = x3 * 3;
}
x7 = ϕ(x4, x5, x6);
print(x7);
```

2.2.2 LowQuadSSA

LowQuadSSA is a lower-level variant of QuadSSA form. It is essentially the same as QuadSSA form except that pointer operations are more exposed. Array access, method invocation, and field access are all replaced with corresponding pointer operations. This representation makes possible pointer transformations which could not be expressed in QuadSSA form. For example, consider the following loop, which can be expressed easily in QuadSSA form:

```
void foo(int arr[]) {
     int max = arr.length;
     for (int i=0; i<max; i++) {
          arr[i] = 42;
     }
}
```

This loop requires two additions per iteration: one to increment the induction variable, and one to perform the array access. Using LowQuadSSA, we could translate it to:

```
void foo(int arr[]) {
     int *max = arr+arr.length;
     for (int *i=arr; i<max; i++) {
          *i = 42;
     }
}
```

which requires only one addition per iteration. This kind of optimization is impossible without the explicit pointers that LowQuadSSA form provides [3].

2.2.3 LowQuadNoSSA

LowQuadNoSSA is not really a "full" codeview. It exists only as a stepping stone in the translation from LowQuadSSA to the Tree form. It uses the same set of operations used by LowQuadSSA, but is not in SSA form. While the ϕ–functions still are present in LowQuadNoSSA code, they are used only to indicate control flow. Their "magic" is gone, replaced by more mundane assignment statements.

2.3 Exposing the Intermediate Representations

The elements of any given IR in the Flex Compiler are accessed through the use of *codeviews* and *dataviews*.

2.3.1 Codeviews

Codeviews are used to expose the elements of a representation for a method. All codeviews provide mechanisms for iterating over their elements, and a pointer to the method they represent. Each intermediate representation defines at least one IR-specific codeview, which optionally contains specialized methods applicable specifically to that IR. For example, tree codeviews have methods for replacing and removing elements from the tree.

All codeviews must implement the `harpoon.ClassFile.HCode` interface, and the elements of a codeview must implement the `harpoon.ClassFile.HCodeElement` interface.

2.3.2 Dataviews

Dataviews are used to expose the elements of an IR for static class data. Unlike codeviews, which represent only method bodies, dataviews can be used to represent many kinds of data: string tables, claz data, etc. Dataviews provide mechanisms for iterating over their elements, and a pointer to the class for which their data applies (dataviews must always be associated with some class).

All dataviews must implement the `harpoon.ClassFile.HData` interface, and the elements of a dataview must implement the `harpoon.ClassFile.HDataElement` interface.

Dataviews generally apply only to IRs which have the ability to directly access memory (i.e., Tree form and lower-level IRs).

2.4 The *Visitor* Design Pattern

Anyone who has studied the landmark text, *Design Patterns*, will be familiar with the so-called *Visitor* pattern. It is used in the Flex Compiler Infrastructure when we wish to add a new operation on some group of classes *without* actually modifying any classes in this group. (The other widely used alternative is to add a new member function to the group of classes to handle the new operation. However, this practice "leads to a system which is hard to understand, maintain, and change." [4]).

While initially confusing, the *Visitor* pattern is really just a fancy way of performing double-dispatch in a single-dispatch language. In a single-dispatch language, the name of the method and the type of the receiver object dictate which operation will fulfill a request. In a double-dispatch language, which operation fulfills a request is based on the name of the operation, and *two* receiver objects: the receiver object, and a parameter[4].

An implementation of a *Visitor* pattern requires two types of participants: a Visitor class, and a set of "Visited" classes. All "Visited" classes must have an `accept()` method, which takes a Visitor as a parameter. The Visitor class must have `visit()` methods for all classes in the set of "Visited" classes:

```
abstract class Visitor {
     public abstract void visit(ElementA elem);
     public abstract void visit(ElementB elem);
}

class ElementA implements VisitorAcceptor {
     ...
     public void accept(Visitor v) { v.visit(this); }
     ...
}

class ElementB implements VisitorAcceptor {
     ...
     public void accept(Visitor v) { v.visit(this); }
     ...
}
```

In this system, a call to the `accept()` method of either the `VisitorAcceptors` first performs single-dispatch on the `VisitorAcceptor` object, and then, from with the body of the `accept()` method, performs another single-dispatch on the `Visitor` object. These two single-dispatch operations result in the desired double-dispatch operation.

In addition to the benefits already discussed, another notable benefit of the *Visitor* pattern is localization of the computation within a single class: each new operation is localized within a Visitor class. The alternative is to have computations spread out across the class hierarchy, which is undesirable [4].

It has been argued that, while the *Visitor* pattern is a useful tool in some object-oriented languages (such as C++ and SmallTalk), it is of less use in Java, where double-dispatch can be simulated more efficiently through the use of the `instanceof` operator. While it is true that the need for the *Visitor* pattern is lessened in Java, the benefits discussed still apply. Furthermore, the `instanceof` operator may thwart various forms of call-structure analysis which will eventually run on the Flex Compiler Infrastructure.

3 Motivation

Appel lists the following as qualities desirable in a good intermediate representation [1]:

1) *It must be convenient for the semantic analysis phase to produce.*

2) *It must be convenient to translate into real machine language, for all the desired target machines.*

3) *Each construct must have a clear and simple meaning, so that optimizing transformations that rewrite the intermediate representation can be easily specified and implemented.*

A tree-based representation satisfies all of these criteria.

It is a straightforward (if tedious) task to implement a translation from LowQuad-NoSSA form to tree form. This can be (mostly) done via a simple mapping that maps quads to small groups of Tree instructions[1], thereby satisfying the first criterion.

It is the second criterion that makes a tree-based representation truly worthwhile for a low-level IR, and this is where Tree form shines. Sethi and Ullman described a mechanism by which a machine-code generator for a tree-based representation could be automatically constructed from a high-level description of the target architecture using Sethi-Ullman numbers [5]. This makes translation to machine code just about as easy as it can possibly get. Nowadays, there are other algorithms with similar properties for modern compilers to choose from. The Flex compiler uses Maximal Munch, which finds an opti-

[1] Translation to Tree form is covered in much greater detail later in the paper.

mal (but not optimum) tiling of the IR tree. Further discussion of instruction selection is beyond the scope of this paper. Consult [1] for more information.

As for the third criterion, the tree-based IR used by Flex has been constructed to be as simple and as orthogonal as possible, making optimizing transformations easy to implement.

The efficacy with which a tree-based representation satisfies these criteria makes it an ideal representation for a low-level IR.

4 Specification of Tree Form

The Tree form used by the Flex compiler owes the core of its design to the tree-based representation used by Appel's *Modern Compiler Implementation in Java*[1]. Readers working with the Flex compiler's Tree form may find Chapter 7 of this book (*Translation to Intermediate Code*) to be a useful reference.

4.1 Elements

The elements which comprise the tree form are divided into two categories: expressions and statements. Expressions represent the computation of some value, and can produce side effects. Statements do not evaluate to a value -- they perform side effects and control flow.

All elements of tree form must extend the abstract class `harpoon.IR.Tree.Tree`. Furthermore, as long as statements and expression form an exhaustive set of all tree elements, each element must extend one of the following two abstract subclasses of `harpoon.IR.Tree.Tree`: `harpoon.IR.Tree.Stm` for statements, `harpoon.IR.Tree.Exp` for expressions.

Each element of tree form is constructed from some set of values, which bestow a unique meaning upon that tree element. These values are referred to as the element's *subexpressions*. Subexpressions can be either literal values, or other tree elements. The ability to nest both expressions and statements ultimately allows us to represent entire methods with one large tree.

The following two sections comprise a "catalog" of the elements of Tree form. Each tree element discussed in the these sections is implemented by a class of the same name as the element itself, in package `harpoon.IR.Tree`.

4.1.1 Expressions

BINOP -- an expression which evaluates to the application of some binary operator to a pair of subexpressions.

BINOP(left, right, op)	evaluates to	**left op right**

CONST -- an expression which evaluates to some constant value. CONSTs represent numeric types only. More complex constant data can be represented with a combination of NAME and DATA trees. The one exception to this rule is the constant `null`, which can be represented by CONST by using its 2-argument constructor. One could conceivably argue that the null constant could be conveniently represented as the integer 0, thereby obviating the need for a "special" null constant. However, some platforms use non-zero values to represent null, which prevents the use of a specific value for the null constant at the tree level.

CONST(4)	evaluates to	**the integer constant "4"**
CONST(-1.2f)	evaluates to	**the floating point constant "-1.2"**
CONST()	evaluates to	**the constant null**

ESEQ -- an expression consisting of a statement, *stm*, and an expression, *exp*, which evaluates to the evaluation of *exp* subsequent to evaluating *stm* for side effects. ESEQs are a bit of a special case in that they are the only *expressions* which can contain statements as subexpressions[1]. As a result, the generic `build()` and `kids()` methods, which return an expression list of children, are not applicable to them. Furthermore, nesting statements within an expression makes control-flow and dataflow analyses exceedingly difficult, as individual expressions may contain control flow [1]. In general, analyses invoked upon

[1] SEQ statements share this property.

tree form will assume the absence of ESEQ nodes (This is referred to as *canonical* form, and is covered later in the paper).

```
ESEQ(                         evaluates to    the integer constant “5”.
        MOVE(
                TEMP(t),
                CONST(5)),
        TEMP(t))
```

MEM -- a generic memory access expression with a single subexpression, *exp,* where *exp* represents an address in memory. The meaning of this tree expression changes depending upon its context within the tree form. When used as the destination operand of a MOVE or a CALL instruction, MEM represents a store to *exp.* When used in any other context, MEM evaluates to the contents of memory at *exp.*

```
ESEQ(                         evaluates to    the contents of memory at the
        MOVE(                                 labeled address “adr”.
                TEMP(t),
                MEM(NAME(adr))),
        TEMP(t))

MOVE(                         evaluates to    a store of the constant “5” to
        MEM(CONST(0x1000)),                   address “0x1000”.
        CONST(5))
```

NAME -- an expression with a single subexpression, *label*, which evaluates to a pointer constant that points to the address specified by *label.* NAMEs are roughly equivalent to uses of an assembly language label.

```
NAME(adr)             evaluates to    a pointer constant which points to the
                                      memory at the labeled address “adr”.
```

TEMP -- an expression which stands for a value in a virtual register. Tree form has an infinite number of virtual registers available. The value stored in a TEMP cannot be modified by any memory access instructions, as might be possible at the machine level. (Without this property, dataflow analysis becomes much more difficult).

TEMP(t1)	evaluates to	**the value stored in virtual register "t1".**

UNOP -- an expression which evaluates to the application of some unary operator to a single subexpression.

UNOP(operand, op)	evaluates to	**op operand**

4.1.2 Statements

ALIGN -- a statement with one subexpression, *alignment*. The evaluation of this statement has the effect of aligning the next statement on an *alignment*-byte boundary (with 0 or 1 specifying default alignment).

ALIGN(4)	represents	**a command to align the next statement on a 4-byte boundary.**
ALIGN(0)	represents	**a command to align the next statement on some default boundary.**

CALL -- a statement which represents a method invocation which uses the Flex Compiler's runtime calling convention. A CALL statement has the following subexpressions: *retval*, a temporary in which to store the return value, *retex*, a temporary in which to store thrown exceptions, *func*, a pointer to the function to invoke, *args*, a list of arguments passed to the invoked method, *handler*, a pointer to the exception handling code for this method, and *isTailCall*, a boolean indicating whether the call should be performed as a tail call.

CALL(**TEMP(tRV),** **TEMP(tRX),** **TEMP(tFunc),** **{ TEMP(tArg1), TEMP(tArg2) },** **NAME(handler),** **false)**	represents	**a method invocation which stores the return value in tRV, the exceptional return value in tRX, invokes tFunc, passes as arguments tArg1 and tArg2, uses handler as its exception handler, and is not invoked as a tail call.**

CJUMP -- a statement with 3 subexpressions, *test*, *iftrue*, and *iffalse*, which represents a conditional branch instruction. *test* must be an expression that evaluates to a boolean result, *iftrue* is a label to jump to if *test* is true, and *iffalse* is a label to jump to if *test* is false.

CJUMP(true, labelT, labelF)	represents	**an unconditional branch to "labelT".**
CJUMP(TEMP(t), labelT, labelF)	represents	**a conditional branch to "labelT" based on the value of the test condition, "t".**

DATA -- a statement which reserves (and optionally writes a value to) a location in memory. DATA statements have two subexpressions: *data*, the expression to write to memory, and *initialized*, a literal boolean expression indicating whether the DATA simply *reserves* memory, or actually writes to it. The location reserved is equal to the location of the nearest preceding label, plus the size of the instructions between that label and the DATA statement.

DATA(null, false)	represents	**the reservation of a word of data, without assigning it a value.**
DATA(CONST(5), true)	represents	**the reservation of a word of data, assigning the value of that memory to be "5".**

EXP -- a statement with one subexpression, *exp*. The evaluation of EXP represents the evaluation of *exp* for side effects.

EXP(CONST(0))	represents	**a nop.**
EXP(**ESEQ(** **MOVE(** **TEMP(t1),** **TEMP(t2)),** **t1))**	represents	**the evaluation of the ESEQ's subexpressions.**

JUMP -- a statement with two subexpressions, *exp*, and *targets*, which represents an unconditional computed branch. *exp* represents the address to jump to, and *targets* represents the list of possible destinations of this jump. If *exp* is a literal label, then the list of possible targets would simply be { *exp* }, rendering *targets* redundant. However, *exp* may also be an expression whose value is computed at runtime, in which case *targets* may contain more than one element. For example, "the C-language `switch(i)` statement may be implemented by doing arithmetic on `i`" [1]. In this case, the list of targets makes it possible to perform dataflow analysis on JUMP trees.

JUMP(NAME(L1), { L1 })	represents	**an uncomputed jump to "L1"**
JUMP(**NAME(L1) + TEMP(T1),** **{ L2, L3, L4, L5 }** **)**	represents	**a computed jump to label "L1" plus some offset. Possible targets for the jump are "L2", "L3", "L4", "L5".**

LABEL -- a statement with two subexpressions, *label,* and *exported.* This statement defines the value of *label* to be the current location in instruction memory (much like an assembly language label). Other statements can reference the location specified by *label* through the use of the NAME expression. The *exported* subexpression is a boolean indicating whether the defined label is visible from other classes and the runtime. Once *label* has been defined by a LABEL statement, it should never be redefined!

LABEL(L2, false)	represents	**the definition of label "L2", which is not visible outside of the current class**

METHOD -- a statement which represents the mapping of formal method parameters to a set of temporaries. The subexpressions for a METHOD statement are a list of temporaries. The first temporary represents a pointer to the exception handling code for this method. For non-static methods, the second temporary represents the `this` pointer for the object upon which this method is invoked. The remaining temporaries represent the formal parameters of this method, in the order in which they are declared.

METHOD(t1, t2, t3, t4)	represents	**the binding of "t1" to a pointer to this method's exception handling code, the binding of "t2" to the `this` pointer, and the binding of "t3" and "t4" to the method's formal parameters.**

MOVE -- a statement with two subexpressions, *src* and *dst*, which represents the assignment of *src* to the location represented by *dst. src* can be any tree expression. *dst* must be either a TEMP or a MEM expression. If *dst* is a TEMP(*t1*), the MOVE instruction represents the assignment of *src* to the virtual register *t1*. If *dst* is a MEM(*adr*), then the MOVE instruction represents a store of *src* to the memory location represented by *adr.*

MOVE(TEMP(t1), CONST(3))	represents	**the assignment of the constant value "3" to the virtual register "t1".**
MOVE(**MEM(NAME(adr)),** **CONST(3)** **)**	represents	**a store of the constant value "3" to the labeled address "adr".**

NATIVECALL -- a statement which represents a function call using the standard C calling convention. Despite the name, NATIVECALLs are *not* generally used to imple-

ment native method calls, as these must use the Java calling convention. Rather, they are used to make calls to the runtime system (such as allocating memory).
Note that NATIVECALLs do not throw exceptions, and functions must signal exceptional behavior through an error code. NATIVECALLs have three subexpressions: *retval*, a temporary in which to store the return value, *func*, a pointer to the function to invoke, and *args*, a list of arguments passed to the invoked method.

NATIVECALL(**TEMP(tRV),** **TEMP(tMalloc),** **{ TEMP(tSize) }** **)**	represents	**a call to the native function pointed to by tMalloc. This is one possible way in which a memory allocation function might be called.**

RETURN -- a statement with one subexpression, *retval*, which represents a return from a method body. *retval* is an expression representing the return value of the method.

RETURN(TEMP(t1))	represents	**returning the value stored in virtual register t1 from the current method body.**

SEGMENT -- a statement representing the beginning of a new section of memory, analogous to memory segmentation on UNIX-based systems [6]. Tree form actually has more segment types than would be available on most platforms. Thus, tree elements residing in different segments in a codeview might end up in the same segment at runtime. *However*, all tree elements within a tree SEGMENT are guaranteed to inhabit a contiguous block of memory at runtime. This is clarified by the following example:

```
SEQ(SEGMENT(SEGMENT.CODE)
SEQ(MOVE(TEMP(t1), TEMP(t2))
SEQ(SEGMENT(SEGMENT.DATA)
SEQ(DATA(CONST(5))
SEQ(SEGMENT(SEGMENT.CODE)
SEQ(MOVE(TEMP(t2), TEMP(t3))
SEQ(SEGMENT(SEGMENT.DATA)
SEQ(DATA(CONST(10)))))))
```

would produce the following pseudo-assembly code[1]:

```
.code                       ## start code segment
mov       r1, r2
mov       r2, r3
.data                       ## start data segment
.word     5
.word     10
```

SEQ -- a statement consisting of two subexpression, *left*, and *right*, both of which must be statements. The SEQ represents the evaluation of *left*, followed by the evaluation of *right*. SEQs are unique in that they have no meaning on their own, and exist solely to provide an ordering between other statements. Furthermore, they are the only kind of *statement* which has statements as subexpressions[2]. As a result, the generic `build()` and `kids()` methods, which return an expression list of children, are not applicable to them. Any code that relies on these methods must include a special case to exclude SEQs.

SEQ(s1, SEQ(SEQ(s2, s3), s4)) represents **the evaluation of statements s1, s2, s3, s4, in that order.**

THROW -- a statement with two subexpressions, *handler*, and *retex*, which represents a thrown expression. *handler* represents the location of exception handling code, and *retex* represents the throwable object thrown by this statement. The astute reader may have noticed that was not strictly necessary to include the handler as part of the THROW object's representation. However, this information makes the exception handling convention more explicit, rather than hiding it amidst some murky fixup code.

THROW(TEMP(t1), TEMP(t2)) represents **throwing the throwable stored in t2, and branching to the handler code at t1.**

[1] The directives used in this example correspond roughly to the GNU as directives[7].

[2] ESEQ *expressions* share this property.

4.2 Properties

Each element of tree form is guaranteed to have certain properties (enforced through interface inheritance).

4.2.1 The UseDef Property

Each element of tree form is guaranteed to possess use/def information: each element knows which `Temps` it defines, and which `Temps` it uses. This is a necessary property for most dataflow analyses, and in general the analyses included in the Flex Compiler Infrastructure operate only upon representations whose elements implement the UseDef property.

Implementation of this property is enforced through the implementation of the `harpoon.IR.Properties.UseDef` interface.

4.2.2 The Typed Property

All expressions in tree form implement the *Typed* property. Many analyses require this information. For example, some algebraic simplifications do not work on non-integral operands, because the limited precision of floating point types could cause the optimized code to yield different results than the original code [8]. An even more important use is the code generation pass of the Flex Compiler, which needs to know the size and representation of expressions to generate correct code.

The Typed property requires expressions to be one of 5 types:

INT:	*32-bit integer*
LONG:	*64-bit integer*
FLOAT:	*32-bit floating point*
DOUBLE:	*64-bit floating point*
POINTER:	*word-sized integer, refers to a memory address*

Implementation of the Typed property is enforced through the implementation of `the harpoon.IR.Tree.Typed` interface.

4.2.3 The PreciselyTyped Property

There are times when the Typed property is not precise enough to define the type of an expression. For example, some native structures could use bitfields rather than the standard java types to save space. To handle these situations, some expressions implement the PreciselyTyped property.

The PreciselyTyped property does not specify a set of predefined types. It requires that expressions export their bitwidth, and whether they are signed. Implementation of the PreciselyTyped property is enforced through the implementation of the `harpoon.IR.Tree.PreciselyTyped` interface.

An item of note is that the `PreciselyTyped` interface extends the `Typed` interface. Thus, any expression which implements `PreciselyTyped` must also implement `Typed`. This may seem unintuitive, as the whole point of the PreciselyTyped property is to be able to type expressions which *cannot* be typed by the Typed property. However, in this case, the Typed property is not used to provide the actual type of the expression. Rather, it is used when a widening conversion must be performed on a PreciselyTyped expression to identify the type of the widened expression. At present, all precise types widen to INT (*32-bit integer*).

4.3 Parents and Siblings

In order to facilitate iteration and manipulation of tree form, each element of tree form maintains a pointer to its parent node, and its right sibling [9]. These can be accessed via the `getParent()` and `getSibling()` methods of `harpoon.IR.Tree.Tree`. The first child of any tree can be accessed through the `getFirstChild()` method.

If the utility of such methods are not obvious at this point, don't worry. Their use will be made plain later in this paper.

5 Using Tree Form Programmatically

5.1 Construction

The preceding sections provide a good conceptual grasp of the tree form. However, a few additional points must be addressed before enough information has been provided to allow for the construction of a valid tree form programmatically.

5.1.1 TreeFactories

The first point is the use of TreeFactories. Similar in nature to the *Abstract Factory* design pattern[4], TreeFactories abstract away many of the details of tree creation. However, rather than using these factories to create trees directly as the *Abstract Factory* pattern would suggest, every element of tree form is required to take a TreeFactory as a parameter in its constructor, which it subsequently passes to its superconstructor. The abstract superclass `harpoon.IR.Tree` then makes use of this TreeFactory to assign the new Tree a unique identifier. Furthermore, this TreeFactory can be used to access method-wide information: i.e., a pointer to the parent codeview, a pointer to the Tree's Frame[1], etc. *All Trees in the same codeview must use the same TreeFactory.*

5.1.2 Source Element

Secondly, each element in the Tree form must be constructed with a *source element.* This source element is supposed to correspond to the `HCodeElement` from which the Tree was translated. It is used to generate the line number and source file information for the Tree.

5.1.3 The Uniqueness Property for Tree Form

Despite its grandiose name, this property is quite simple. It states that each Tree element within a codeview or dataview must be unique. That is, if a Tree element, A, occurs at one context in a Tree, it may not occur in any other context. Relying on this property greatly simplifies both the implementation and use of analyses on Tree form.

One confusion resulting from this property was the over the correct way to assign a value to a temporary, and then access it again later. Clearly we could assign a value to a TEMP object, but how could we reuse it without reusing the TEMP elsewhere in the tree?

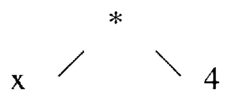
[1] Not covered in this paper, Frames are a construct used by the Flex compiler to carry platform-specific information.

The answer to this question is that each TEMP object has as a subexpression an instance of `harpoon.Temp.Temp`. It is this underlying `Temp` that determines where the result of an assignment is stored. Thus, the solution is to use different TEMP objects, both of which have as their subexpression the same `Temp`.

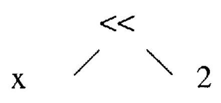
5.1.4 Examples

Example: Given Tree A, construct Tree B

A:



B:



```
BINOP times4ToShift2(BINOP A) {
     TreeFactory tf = A.getFactory();
     CONST two = new CONST(tf, A, 2);
     BINOP B = new BINOP
          (tf, A, Type.INT, Bop.SHL, A.getLeft(), two);
     return B;
}
```

Example: write a function that takes in a expression, *n*, and returns a tree which evaluates to *factorial(n)*.

```
import java.util.LinkedList;

Exp generateFactorial(Exp n) {
     TreeFactory tf = n.getFactory();
     TempFactory tmpF = tf.tempFactory();
     LinkedList stms = new LinkedList();

     // Create virtual registers in which to store temporary
     // computations.
     Temp tResult = new Temp(tmpF);
     Temp tN = new Temp(tmpF);
```

```
// Create symbolic addresses.
Label TEST = new Label();
Label LOOP = new Label();
Label DONE = new Label();

// tN <-- n
stms.add(new MOVE(tf, n,
          new TEMP(tf, n, tN),
          Tree.clone(tf, null, n));

// tResult <-- 1
stms.add(new MOVE(tf, n,
          new TEMP(tf, n, tResult),
          new CONST(tf, n, 1));

// if tN > 0, continue.  Otherwise, goto DONE.
stms.add(new CJUMP(tf, n,
          new BINOP(tf, n, Type.INT, Bop.GT,
               new TEMP(tf, n, tN),
               new CONST(tf, n, 0)),
          LOOP,
          DONE);

stms.add(new LABEL(tf, n, LOOP);

// tResult <-- tResult * tN
stms.add(new MOVE(tf, n,
          new TEMP(tf, n, tResult),
          new BINOP(tf, n, Type.INT, Bop.MUL,
               new TEMP(tf, n, tResult),
               new TEMP(tf, n, tN))));

// tN <-- tN - 1
stms.add(new MOVE(tf, n,
          new TEMP(tf, n, tN),
          new BINOP(tf, n, Type.INT, Bop.ADD,
               new TEMP(tf, n, tN),
               new UNOP(tf, n, Type.INT, Uop.NEG,
                    new CONST(tf, n, 1)))));

// GOTO TEST
stms.add(new JUMP(tf, n, TEST));

// All done!
stms.add(new LABEL(DONE));
```

```
        // Return the result as an expression.
        return new ESEQ
             (tf, n,
             Stm.toStm(stms),
             new TEMP(tf, n,tResult));

}
```

5.2 Modification

One downfall of a tree-based representation is the difficulty of removing or replacing tree nodes. The following are sources of this difficulty:

1) *The parent and sibling pointers of each tree must be maintained correctly.* Failure to maintain these pointers correctly is a grievous error, and potentially hard to debug.

2) *Tree elements do not have a good generic way to access their subexpressions.* Each tree element exposes access to its subexpressions differently. For example, BINOPs provide `getLeft()` and `getRight()` methods, CJUMP has a `getTest()` method, etc. Sometimes this is OK, but oftentimes we do not know the exact type of the expression which we're operating on. In such cases, it is necessary to first determine the type of the expression, and then to call the appropriate accessor methods -- a tedious task at best[1].

3) In rare cases, it may be necessary to replace or remove the root element of a codeview. The root element of a codeview is protected by access modifiers, and even if it weren't, it would be very bad form to be mucking around with the internal representation of a codeview externally. Furthermore, checking for this special case is annoying!

[1] This design was permitted because of the difficulty of constructing a good generic way of accessing tree subexpressions. In fact, a generic access mechanism exists in the form a Tree's `build()` and `kids()` methods, but it is non-intuitive and inefficient.

Because of both the difficulty in modifying Tree form and the demand for such modifications, codeviews which extend harpoon.IR.Tree.Code all inherit methods by which to modify Tree form. These methods hide the difficulties of modifying Tree form from the programmer. This modification interface consists of two methods: one to remove a statement, and one to replace an arbitrary tree with another:

```
public void remove(Stm stm);
public void replace(Tree tOld, Tree tNew);
```

Note that the remove() method operates only on statements, while the replace() method operates on arbitrary trees. The reason for this decision was the difficulty in coming up with a meaningful definition of "removing an expression" from a tree. Simply removing the expression would leave the parent node with a missing child, and the meaning of the node would become undefined. Rather than imposing some other semantics upon the removal of expressions, the easier solution seemed to be to simply disallow this operation.

5.2.1 Examples

Example: Combining Constants

The following code iterates over all of the elements in a tree codeview, and evaluates binary operators with constant subexpressions and replaces them with new constant values. **Note:** while this code is instructive, it could never be used in a real simplification pass, as it behaves incorrectly for division-by-0.

```
combineCONSTs(Code code) {
   for (TreeWalker i=code.treeWalker(); i.hasNext();) {
      Tree next = (Tree)i.next();
      if (next.kind() == TreeKind.BINOP) {
         BINOP b = (BINOP)next;
         if ((b.getLeft().kind()==TreeKind.CONST) &&
              (b.getRight().kind()==TreeKind.CONST) {
            CONST newVal = BINOP.evalValue
               (b.getFactory(), b.op, b.optype,
                  b.getLeft(), b.getRight());
            code.replace(b, newVal);
         }
      }
   }
}
```

5.3 Iteration

The sample code from the previous section contained a forward reference to the topic of this section: mechanisms for iterating over tree form.

As previously discussed, each codeview provides a mechanism for iterating over its elements. Two qualities which are desirable for codeview iterators are *robustness* and *extensibility.* Robustness in this context refers to an iterator's ability to gracefully handle changes to the codeview over which it iterates. Extensibility refers to the ease of which the behavior of the iterator can be customized.

5.3.1 First Attempt

The first implementation of iterators for the Tree form was written *before* each tree element had a sibling and a parent pointer. It used a stack to maintain the state of the traversal. Unfortunately, it is not possible to implement a robust traversal using this scheme. A change to the underlying Tree form invalidates the entire stack, and without sibling and parent pointers, the iterator cannot recover gracefully.

This is clearly less than robust behavior in the face of modification, and this implementation did not even attempt to make a provision for extensibility.

5.3.2 Second Attempt

The new iterators for tree form do much to remedy the problems of the first ones. They are based on the iterators found in the w3c's Document Object Model [9]. The algorithm used to iterate over the tree form is:

```
NextNode() {
     if current node has any children:
          return first child;
     else:
          Let TMP be the current node.
          while (true):
               if TMP is null, return null.
               else:
                    if TMP has any siblings:
                         return next sibling.
                    else:
                         Let TMP be TMP's parent.
}
```

To see how robust this implementation is, we must discover how it behaves when we modify the tree form during iteration. Recall that the operations allowed on tree form are *replace*, and *remove*. Let's enumerate the possible modifications one could make on the tree form and see how the new iterator behaves:

1) *Replacing or removing a node before the current node, but not one which causes the removal of the current node from the codeview.*

The iterator will continue to iterate over the rest of the tree, ignoring this change to the tree form. This is acceptable, intuitive behavior, and is consistent with the behavior of iterators in Flex's quadruple-based representations.

2) *Replacing a node after the current node.*

The iterator will iterate the replaced node instead of the original one. Again, intuitive behavior.

3) *Removing a node after the current node.*

The iterator will skip this node in its iteration. This behavior is also intuitive.

4) *Removing the current node, or a previous one which is the root of a subtree in which the current node resides.*

Using the algorithm described, the tree iterator will continue to iterate over the subtree of the removed node, but will not iterate the rest of the tree.
This behavior is considered desirable by the DOM specification[9].
However, it is probably not what is wanted here. If a node is removed from an IR, the preferred behavior would be to continue iterating from the next node still in the tree.

5) *Replacing the current node, or a previous one which is the root of a subtree in which the current one resides.*

This is the same state of affairs as in the previous paragraph. The algorithm described would result in the iterator continuing to iterate over the replaced node's subtree, but not the rest of the tree. In this case, the preferred behavior would be to continue iteration from the *new* node.

5.3.3 Third Attempt

Our second attempt at robust iterators was a great improvement, but it still exhibited undesirable behavior in certain cases. The solution to this problem is based on the `java.util.Iterator` interface defined in the Java Collections Hierarchy[10].

In Java, `Iterators` export a mechanism through which the underlying Collection over which they iterate may be modified:

```
public interface Iterator {
     public boolean hasNext();
     public Object next();

     /**
      * Used to remove the last element iterated from
      * from this Iterator's underlying Collection. */
     public void remove();
}
```

The behavior of the `Iterator` is undefined if the underlying Collection is modified while iteration is in progress through any other mechanism than the `Iterator`'s `remove()` method.

If we enforce a similar constraint for our Tree Iterators, we can remedy the problems of our previous attempts. When the Tree Iterator is asked to remove or replace the current node in the iteration (or one of its ancestors), it is able to update its internal state appropriately so as to exhibit the desired behavior.

We have at last designed a suitably robust implementation of iterators for the Tree form. However, we have still not provided a mechanism for extensibility. For this, we look again to the w3c's Document Object Model. Therein is described a mechanism for customizing iteration called "filtering". Before each node is returned by an iterator, a filter is applied to determine whether or not the node should be returned, and whether its subtree should be enumerated.

This system is specified in the Flex Infrastructure in the `harpoon.Util.IteratorFilter` interface:

```
public interface IteratorFilter {
     public int filter(HCodeElement node);
}
```

This interface is made more specific by the `harpoon.IR.Tree.TreeIteratorFilter` subinterface, which provides a more concrete specification and some tree-specific constants:

```
public interface TreeIteratorFilter extends
     IteratorFilter {

     public static final int ACCEPT = 0;
     public static final int REJECT = 1;
     public static final int SKIP = 2;

     /**
      * Returns ACCEPT if the tree should be iterated,
      * returns REJECT if the tree should not be
      * iterated, but its subtree may be.  Returns
      * SKIP if this node should not be iterated, and
      * its subtree should be "skipped over".
      */
     public int filter(HCodeElement tree);
}
```

While the iteration process would be fully customizable with only ACCEPT and REJECT return values, the presence of a SKIP value allows for much more efficient iterations. For example, the following filter would allow for the efficient iteration of allow of the statements in a tree codeview:

Example: Statement filter

```
public class StatementFilter implements
     TreeIteratorFilter {

     public int filter(HCodeElement hce) {
          Tree tree = (Tree)hce;
          if (tree instanceof Stm) {
               return ACCEPT;
          }
          else if (tree instanceof Exp) {
               return SKIP;
          }
          else {
               throw new Error("What the?");
          }
     }
}
```

The final implementation of the tree iterators exists in the `harpoon.IR.Tree.TreeWalker` class.

6 Dataviews

Tree form is the first IR which is used to actually represent static data explicitly. It is in tree form that such structures as the *virtual method table* and the *reflection tables* are constructed. As with code, it is easier and more robust to model *data* in some architecture independent format, rather than having to implement data layouts for all supported platforms.

There is a lot of data that needs to be laid out in memory for a Java program to function correctly. This includes:

1) *Claz structures*
2) *Tables for reflection*
3) *String tables*
4) *Static fields*
5) *Static initializers*

6.1 Claz Structures

Each Java class has a *claz structure* associated with it. The Flex claz structure has the following specification:

Offset from claz pointer	Data
-P*N	Nth Interface Method
...	...
-2P	2nd Interface Method
-P	1st Interface Method
0	Claz Info Pointer
P	Component Type Claz Info Pointer
2P	Interface List Pointer
3P	Class Size
3P + W	Class Depth (D)
3P + 2W	`java.lang.Object` Claz Info Pointer
4P + 2W	(D-1)th Superclass Claz Info Pointer
...	...
(3+D)P + 2W	This Class's Claz Info Pointer
(4+D)P + 2W	Pointer to 1st Class Method
(5+D)P + 2W	Pointer to 2nd Class Method
...	...
(3+M+D)P + 2W	Pointer to Mth Class Method

Figure 3: The Flex Claz Structure[1]

In the above diagram, *interface methods* refer to methods specified in interfaces implemented by the class. An interface's methods are assigned an index by the Flex compiler, and occupy that index in *every* claz structure in which they exist. Flex employs a graph-coloring algorithm to minimize the space required by this scheme. The *Claz Info Pointer* is a pointer to the structure itself. For claz's representing arrays, the *Component Type Claz Info Pointer* is a pointer to the array's component type. For non-array types, this field is

[1] In this, and in all Claz Structure figures: W = word size, P = pointer size. Specific to this figure, N = number of interface methods, M = number of class methods, D = class depth

NULL. The *Interface List Pointer* is a pointer to a NULL-terminated list of interfaces implemented by the class. *Class size* is the size of an instance of this class. *Class depth* is this class’s depth in the class hierarchy. For a class depth of D, the following D fields represent the classes from which this class inherits, ordered by class depth (lowest depth first). Finally, the remaining fields form a virtual method table of all of this class’s methods.

The Flex claz structure is designed to be both compact and efficient. While it may seem daunting at first, the layout of the claz structure is actually quite simple once the logic behind it is known. This structure is accessed at runtime to allow a Java program to:

1) *Perform virtual method invocations on both class and interface methods*

First and foremost, the claz structure is a virtual method table. Dynamically dispatched calls rely on the claz structure to operate correctly. A method invocation using this claz structure to perform dynamic dispatch might be compiled like so:

Java:

```
Object ref = new Foo();
ref.bar();
```

Pseudo-compiled code:

```
Pointer ref = /* code to create new Foo object */;
// Get a pointer to ref's claz structure
Pointer claz_ptr = *(ref+offset_of_claz_ptr);
// Get a pointer to this class's bar() method
Pointer meth_ptr =
     claz_ptr + offset_of_1st_method + index_of_bar;
// Invoke bar()
(*meth_ptr)();
```

2) *Perform instanceof checks on classes, interfaces, and arrays*

A strong point of this claz structure is the speed with which it performs instanceof checks on classes. Because Java allows only single inheritance from classes, a class’s entire ancestry can be indexed by *class depth.* For instance, given the following class definitions:

```
class A { }
class B extends A { }
class C extends B { }
```

The claz data would for class C would contain the following inheritance information:

Offset from Object Reference	Data
...	...
class depth offset (CDO)	4 (depth of class C in the inheritance hierarchy)
CDO + W	`java.lang.Object` (ancestor at depth=1)
CDO + W + P	A (ancestor at depth=2)
CDO + W + 2P	B (ancestor at depth=3)
CDO + W + 3P	C (the class itself)
...	...

Figure 4: Inheritance Information in the Claz Structure

This would allow for an `instanceof` check to be compiled like so:

Java:

```
    boolean foo(java.lang.Object obj) {
        return obj instanceof java.lang.String;
    }
```

Pseudo-compiled code:

```
boolean foo(Pointer obj) {
     // Get a pointer to the claz structure.
     Pointer claz_ptr = *(obj+offset_of_claz_ptr);
     // If "obj" is an instance of String, than String
     // MUST be its ancestor at depth=2 (String's depth
     // in the class hierarchy).
     Pointer class_data_ptr =
          claz_ptr + class_depth_offset +
               class_depth_of(java.lang.String);
     return *class_data_ptr == java.lang.String;
}
```

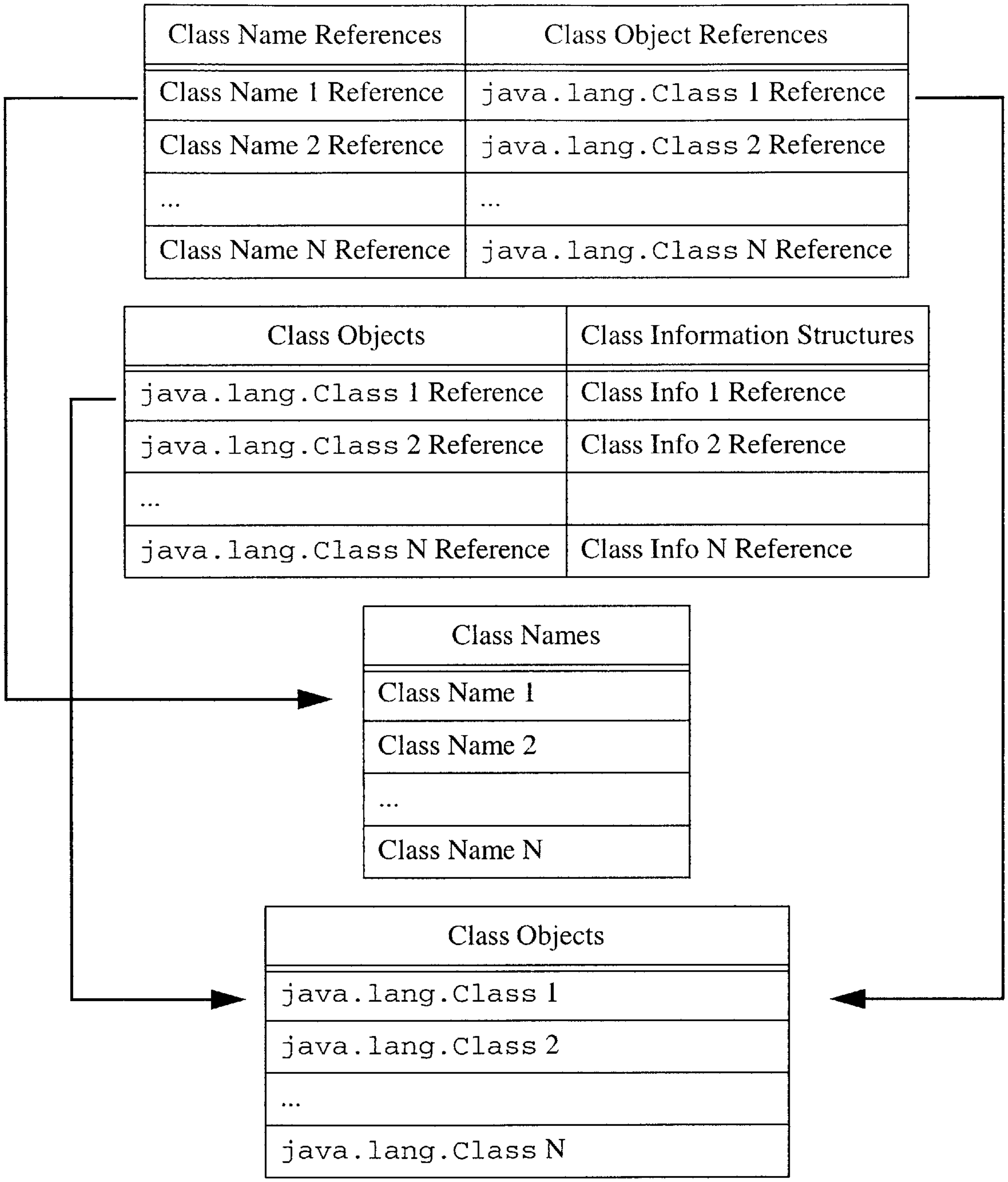
Performing instanceof checks on interfaces is not nearly as efficient, as the entire NULL-terminated interface list must be searched. However, interface instanceofs are usually much rarer, and more importantly, they are not used in the Java class libraries.

6.2 Reflection Tables

Tree form also creates several tables to support java's *reflection* mechanism. The first is a mapping of all class names to their corresponding `java.lang.Class` objects. The table is sorted by class name to allow for log(n) lookup time. The second is a mapping of all `java.lang.Class` objects to class information structures (not Claz structures). [1] This table is sorted by `java.lang.Class` object address to allow for log(n) lookup time. Because of their use in indexing the reflection tables, `java.lang.Class` objects are guaranteed by the runtime system to be non-relocatable[3]. The third reflection table consists of the actual UTF-8 strings encoding the class names, and the fourth consists of the actual `java.lang.Class` objects. (These `Class` objects don't contain any data, they are used merely as keys to index the second reflection table.

The following diagram shows these relationships:

[1] The Flex compiler performs *whole-program analysis*. That is, every class used by a program is recompiled and analyzed on each compile. This absence of dynamic linking allows us to actually enumerate a complete list of classes here.



Class Name References	Class Object References
Class Name 1 Reference	`java.lang.Class` 1 Reference
Class Name 2 Reference	`java.lang.Class` 2 Reference
...	...
Class Name N Reference	`java.lang.Class` N Reference

Class Objects	Class Information Structures
`java.lang.Class` 1 Reference	Class Info 1 Reference
`java.lang.Class` 2 Reference	Class Info 2 Reference
...	
`java.lang.Class` N Reference	Class Info N Reference

Class Names
Class Name 1
Class Name 2
...
Class Name N

Class Objects
`java.lang.Class` 1
`java.lang.Class` 2
...
`java.lang.Class` N

Figure 5: Reflection Tables

Finally, the aforementioned class information structures contain the class name, a pointer to the class's claz structure, and a mapping of member signatures to method and field offsets. This is the table most people envision when they think of Java's reflection mechanism.

6.3 String Constants

For efficiency, all string constants can be pre-allocated by the compiler. This saves both computation time, and memory. Computation time is saved because we 1) avoid a call to the runtime system to allocate memory and 2) avoid a call to `java.lang.String`'s constructor. Memory is saved because copies of the same String constant need to be allocated only once. Even if this scheme *didn't* save resources, we would have little choice in the matter, as the *Java Language Specification* requires that String literals be "interned"[10] so that for any two String constants s1 and s2:

```
s1.equals(s2)        iff         s1 == s2
```

One potential downfall of this approach is that performance degradation could be observed in cases where large String constants were declared, but never used. If this unlikely problem were actually the cause of performance degradation, it could easily be remedied by reading the largest Strings from a file as needed. In a research compiler infrastructure, that is a sufficient solution. Regardless, it is unlikely that this feature would become a source of performance degradation.

The one difficulty in laying out String constants is that the entire `String` object structure must be created, not simply the character array representing the value of the String constant. Fortunately, the Flex object layout is not complex:

Offset from Object reference	Data
0	Pointer to claz structure
P	Hashcode
W + P	Field 1
W + P + sizeof(field 1)	Field 2
...	...
W + P + sizeof(fields 1 through n-1)	Field n

Figure 6: Flex Object Layout[1]

Using the above specification of object layout, we can *almost* determine the layout of `String` objects. The one problem is that the "value" field of the `String` class is a character array, so we also need to know the Flex specification for array layout:

Offset from array reference	Data
0	Point to claz structure
P	Hashcode
P + W	Array length
P + 2W	Element 0
P + 2W + sizeof(component type)	Element 1
...	...

Figure 7: Flex Array Layout

[1] Notably absent from this Object layout is a "mutex" field. At present, a synchronization mechanism has not been devised for the Flex compiler. Once such a mechanism is decided upon, a lock will be added to the object structure (probably by stealing bits from the hashcode and/or the claz pointer).

Using the above specifications, laying out properly initialized String objects in memory is as simple as looking at the source code for the java.lang.String class. The fields we need to represent are:

```
// A pointer to the character array representing the
// value of this String
char[] value;

// The offset from the first character in "value" at
// which this String begins.
int offset;

// The length of this String, in characters.
int length;
```

We are, at last, ready to construct properly initialized String objects, simply by following the above specifications. For example, the String "test" would be represented through the following data structures:

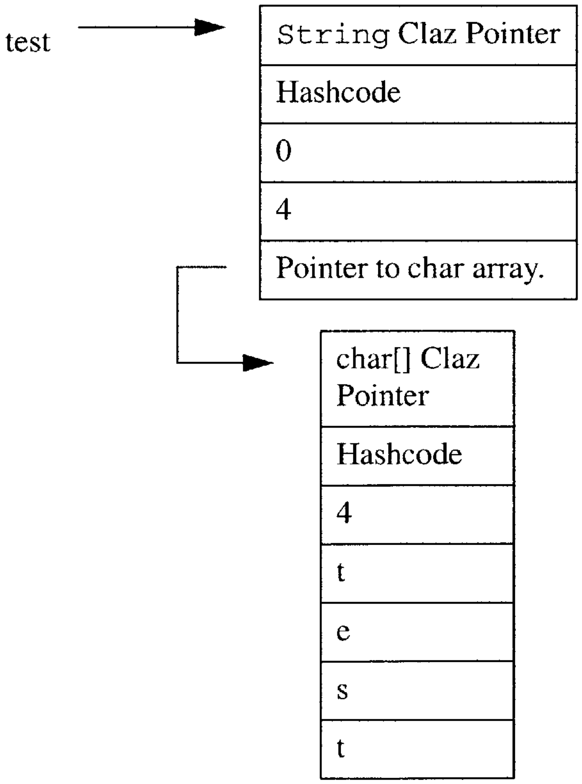


Figure 8: The String "test"

6.4 Static Fields

As with all static class data, static fields also need to be initialized at some globally accessible location. The details of static field layout are unremarkable. The details of note are:

1) *Fields are sorted by size to save space.*
2) *Pointers are laid out before primitive types to simplify garbage collection.*

Otherwise, static field layout is nothing more than a list, with each field accessed through a symbolic label.

6.5 Static Initializers

This dataview outputs a table listing the static initializers which must be executed, in the correct dependency order [3]. As with reflection data, our ability to lay this data out statically is a result of Flex's *whole-program analysis.* Performing whole-program analysis lets us know at compile-time exactly which classes are to be loaded during the programs execution. Flex then takes this list of classes, and arranges to call each of their static initializers, in the correct order.

7 Constructing Tree Form from LowQuadSSA

In the introduction of this paper, the IRs of the Flex Compiler were enumerated. The careful reader may note that it was LowQuadNoSSA that preceded Tree form, not LowQuadSSA. The reason that this section discusses the transformation of LowQuadSSA to Tree form is that LowQuadNoSSA exists entirely as an intermediate phase of the translation from LowQuadSSA. LowQuadSSA is the first "true" codeview which precedes the Tree form.

The translation from LowQuadSSA to Tree has three phases:

1) LowQuadSSA is converted to LowQuadNoSSA.
2) Labels are added to the destination of every branch.
3) Quads are mapped to Trees.

7.1 LowQuadSSA to LowQuadNoSSA

As mentioned earlier, LowQuadNoSSA is simply LowQuadSSA form, with all "magical" operators removed. This includes the magical ϕ–functions, and the not-magical, but wasteful σ–functions, which would otherwise cause unnecessary copy statements to abound in the Tree form.

The transformation from LowQuadSSA to LowQuadNoSSA is implemented in the `harpoon.IR.Quads.ToNoSSA` class[1]. At its core, the transformation consists of the execution of two algorithms: one to remove ϕ's, one to remove σ's.

7.1.1 Removing the σ–functions

First, "*removing*" is the wrong word. This phase does not actually *remove* the σ–functions -- they still exist to indicate control flow. Rather, the n-ary copy operation is removed from each σ–function.

The removal of these copy operations is implemented in the `harpoon.IR.Quads.ToNoSSA.SIGMAVisitor` class. This removal is accomplished through the following algorithm:

```
RemoveSIGMAs(QuadGraph g) {
     Let M be an empty Map
     Foreach element q in g:
          If q is a SIGMA function:
               Foreach assign s(t1,t2...tn-1)=tn in q:
                    add{(tn,t1),(tn,t2)...(tn,tn-1)}
                         to M.
     Foreach element q in g:
          For each Temp t in q:
               tOld = t
               while M.containsKey(t):
                    t = M.get(t)
               replace tOld with t in q
}
```

[1] This class could also be used to convert from QuadSSA to QuadNoSSA, if this ever became necessary.

In English, this algorithm consists of two passes. The first pass traverses the quad graph, and records all assignments performed at SIGMA nodes. The second pass applies these assignments to each temporary in the quad graph:

Example: Removing SIMGAs from a code fragment.

Before

```
...
σ(i1, i2) = i;
if (i > 1) {
     i3 = i1
}
else {
     i4 = i2 + 1;
}
i5 = ϕ(i3, i4);
print(i5);
```

After

```
...
σ;             // σ is still here, but does nothing.
if (i > 1) {
     i3 = i
}
else {
     i4 = i + 1;
}
i5 = ϕ(i3, i4);
print(i5);
```

This algorithm has the advantage of being extremely simple. It is easy to understand and implement. The one obvious disadvantage is that it requires two iterations over the quad graph, whereas conceivably this transformation could be completed in just one. The reasons that this optimization was not implemented was:

1) The performance hit observed for this extra iteration was negligible compared to the overall running time of the compiler.

2) This optimization would require a reverse-postorder quad iterator which, is not presently part of the interface to quad form.

3) The current transformation is believed to work reliably.

Because the transformation already runs at a reasonable speed, we cannot at this time justify adding a small optimization which could be a potential source of bugs in the Flex Compiler Infrastructure. However, as the compiler matures, performance tuning may become more of an issue. At such a time, this transformation could certainly be the target of optimization efforts.

7.1.2 Removing the ϕ–functions

Again, "*removing*" is the wrong word, as the ϕ–functions must remain to indicate control flow. Rather, the magical n-ary selection operator is removed, reducing the ϕ–function to a mere placeholder.

The removal of this selection operation is implemented in the `harpoon.IR.Quads.ToNoSSA.PHIVisitor` class. This removal is accomplished through the following algorithm:

```
RemovePHIs(QuadGraph g) {
  Foreach element q in g:
    if q is a PHI function:
      Foreach selection, tn=f(t0,t1...tn-1), in q:
        For i=0 to n-1
          Let qPrev be the ith predecessor of q.
          Let m be MOVE(tn, ti)
          Insert m between qPrev and q.
      Let q' be a PHI node equivalent to
        q, except that it performs no
        selection operations.
      Replace q with q'.
}
```

Example:

Before

```
σ(i1, i2) = i;
if (i > 1) {
     i3 = i1
}
else {
     i4 = i2 + 1;
}
i5 = ϕ(i3, i4);
print(i5);
```

After

```
σ(i1, i2) = i;
if (i > 1) {
     i3 = i1
     i5 = i3;
}
else {
     i4 = i2 + 1;
     i5 = i4;
}
ϕ;               // ϕ is still here, but does nothing
print(i5);
```

Recall that the ϕ–function magically selects a source operand based on which path of execution was taken. With the restrictions of SSA lifted, this algorithm makes that operation explicit by inserting a MOVE instruction after each direct predecessor of the ϕ–function. Each MOVE instruction, *m*, selects its execution path as the correct one by performing the assignments the ϕ–function would have performed once it determined that *m*'s execution path was the correct one. Since *m* will only get called if its path is taken, this is a correct translation of the ϕ–function.

Like the algorithm to remove σ–functions, this algorithm has the advantage of simplicity -- it is easy to understand and implement. Furthermore, it does not share the disadvantage of the RemoveSIGMAs algorithm, in that it requires only one traversal of the quad graph.

7.1.3 Possible Improvements

In addition to reducing the RemoveSIGMAs algorithm to one pass, it is possible that the entire conversion to NoSSA form could be implemented in one pass. This would be somewhat tricky, as the naive one-pass implementation of RemoveSIGMAs requires a reverse-postorder traversal of the quad graph, and the RemovePHIs implementation involves "backtracking" from the PHIs encountered by the algorithm to insert MOVE statements. It is therefore likely that this improvement would require a much more complex implementation.

Another potential improvement would be the modification of the LowQuadNoSSA codeview to ignore the selection operators in p-functions and the n-ary assignment operators in σ–functions. This would obviate the need to explicitly remove them from the LowQuadSSA codeview, and could be a far less buggy optimization than the one-pass optimization.

Regardless, both of these proposed optimizations would likely yield only small gains compared to the overall running time of the compiler. Thus it is advisable that other phases of compilation be optimized first, and only then that attention be directed towards this transformation.

7.2 Labeling Branch Destinations

In Flex's quad-based representations, control flow is an implicit property of the quadruples. Each quad has a set of "next" and "prev" edges, which connect it to other quads to form a quad graph. This is not an acceptable low-level representation, as hardware cannot easily simulate these "edges". Explicit branch instructions are necessary. Therefore, wherever two adjacent trees cannot reach each other by fall-through execution, we must insert a branch instruction. *However*, for each branch instruction we wish to insert, we need to have some label representing the destination of the branch. Unfortunately, quads don't use labels -- they don't need to. Therefore, wherever Tree form requires an explicit branch, we need to insert a label at the branch destination.

We must therefore determine exactly which quads in the quad graph must be labeled. A little introspection reveals the answer:

1) Any node following a SIGMA node.
2) Any PHI node.

Unfortunately, it is difficult to see if this set of quads works in all cases. The following proof shows that this set of quads is both correct, and exhaustive:

Theorem: Let Q be the set of nodes in a quad graph G.
Let P be the set of PHI nodes in G.
Let S be the set of nodes following SIGMA nodes in G.
\ The nodes $P \cup S$ should be labeled *and* the nodes in $Q - (P \cup S)$ need not be labeled.

Assumption: If two nodes reside in the same basic block, there are no branch instructions between the two nodes.

Proof:
Step 1: The nodes in P should be labeled.
If a node, *q*, is in P, then it has more than one predecessor.
At most one predecessor of *q* can reach *q* by fall-through execution. All others must reach it by a branch.
If *q* is in P, *q* should be labeled.

Step 2: The nodes in S should be labeled.
If a node, *q*, is in S, then it is preceded by a SIGMA node, *s*.
Quad form's SIGMAs can be translated to either tree CALLs or tree CJUMPs.
Tree CJUMPs require that both the true and the false branch be labeled.
Tree CALLs require that the exception-handling code be labeled. While they don't require that the successor in the normal execution path be labeled, the translator often places the handler code directly after a method call, thereby requiring a JUMP to reach this successor. Given this behavior of the translator, both successors of a CALL should be labeled.
All successors of a SIGMA should must be labeled.
If *q* is in S, it should be labeled.

Step 3: The nodes in $N = Q - (P \cup S)$ need not be labeled.
Let q be a node in N.
q must have 1 predecessor, or else it is in P.
q's predecessor must have 1 successor, or else q is in S.
q and q's predecessor must be in the same basic block.
Since there are no branch instructions between q's predecessor and q, q does not need to be labeled.

Theorem Proved.

Now that the set of nodes to map to labels has been determined, the actual labeling algorithm follows trivially:

```
LabelBranchDestinations(QuadGraph g) {
     Foreach element q in g:
          if q is a PHI node and is not marked
               as replaced:
               Mark q as replaced.
               Replace q with an equivalent LABEL node.
          if q is a SIGMA node:
               Foreach successor s of q:
                    if s is not marked as replaced:
                         Mark s as replaced.
                         Insert a LABEL node between
                              q and s.
}
```

The one subtlety of this algorithm is the reason that PHI nodes should be *replaced* by LABELs, while all other nodes require that a LABEL be *inserted.* The reason for this is the LABEL extends PHI, and it can therefore be used in place of a PHI without disruption. The same does not apply for other nodes, which may perform some critical function that cannot be simulated by a LABEL node.

7.3 Map Quads to Trees

The remainder of the translation from LowQuadSSA to Tree form is relatively straightforward. Essentially, the last phase of translation is a simple mapping: quads are mapped directly to some small set of trees which best approximate the meaning of the quad. Most quads can be mapped very simply onto trees. For example, the MOVE quad is translated like so:

Example: Translating the MOVE quad.

```
public Tree translate(harpoon.IR.Quads.MOVE q) {

     return new harpoon.IR.Tree.MOVE
          (tf, q,
           new TEMP(tf, q, q.dst()),
           new TEMP(tf, q, t.src())
          );
}
```

The `LABEL` quad has an equally simple translation:

Example: Translating the LABEL quad.

```
public Tree translate(harpoon.IR.Quads.LABEL q) {
     return new harpoon.IR.Tree.LABEL
          (tf, q, q.label, q.exported);
}
```

It is not my intent to present a "catalog" of these trivial mappings, as they can easily be discovered simply by examining the `harpoon.IR.Tree.ToTree` class. However, there are some subtleties which arise in this approach, which should be clarified:

1) *Ensuring that the correct control flow is preserved when laying out the tree form is more difficult than it seems.*

Maintaining correct control flow in quad form is easy, as quads are designed to be members of a control flow graph, and can be connected to each other simply through the assignment of edges. Elements of tree form are not designed to be members of a graph structure, and cannot be connected in that fashion -- a control flow edge exists between two trees *t1* and *t2* only if a) *t1* directly precedes *t2* or b) *t1* branches to *t2*.

Now our problem is evident: the predecessors of join points in quad form didn't need to worry about explicit control flow: they just connected themselves to the join point with an edge. This isn't possible in tree form: all but one predecessor of a join point must be connected through a branch instruction.

Fortunately, the solution to this problem is not difficult. The tree translator simply iterates over the quad graph is depth first order. If it encounters a node which it has seen before, it assumes this is a join point, and inserts an explicit JMP[1] instruction between the join point and its predecessor.

One concern of this algorithm is that it may create more jump instructions than is strictly necessary through poor code placement. This is acceptable in this case, as a code placement algorithm is usually run directly before the code-generation phase to minimize the number of these unnecessary jump instructions[1].

2) *Some quads, when translated to Tree form, need access to native information not specified in the tree form itself.*

Such native information might include: claz layout, object layout, access to runtime system functions, etc. The solution is to parameterize the tree translator with a `TreeBuilder` class. Any tree element which requires runtime information is created using this builder. This flexible design allows us to translate to an entirely different runtime system just by using a different `TreeBuilder`.

It is important to note that, while the tree translator must translate to low-level code based on a certain data format, dataviews must also be created which can load data of this format into memory when the compiled program executes. Thus, the tree translator and all of the supported dataviews *must* use the same `TreeBuilder` in the course of any compile.

[1] Although there are no explicit jump instructions in quad form, a specialized one was defined within the tree translator just for this purpose.

3) *Some quads translate differently based on their context.*

Some tree elements are best translated in different ways based on their context within a program. For example, consider the translation of the constant “0”. When used as the right hand side of an assignment, it does simply translate best as a the constant 0. However, when used as the conditional of CJUMP instruction, the whole statement might best translate to an unconditional jump to the false branch.

For tree elements which may translate differently in different contexts, we have created the abstract `harpoon.IR.Tree.Translation.Exp` class. When a class inherits from Translation.Exp, it is saying that it may have a different translations based on its context. The three allowed contexts are:

1) *Simple expression*

The Tree element is used as a simple expression. In general, only tree expressions can be simple expressions -- there is no default conversion from statement to expression. Note that this is only the default implementation, and subclasses of `Translation.Exp` may override this behavior as appropriate.

2) *Simple statements*

The Tree element is used as a simple statement. Both statements and expressions may be used as simple statements. The default conversion from expression to statement is to wrap an EXP around the expression.

3) *Conditional branches*

The Tree element is used as a conditional branch instruction. Only expressions and conditional branches have default conversions to conditional branch instructions. The default conversion of expression to conditional branch simply uses the expression as a test as part of a conditional. Eventually, the default conversion may account for cases like the one de

scribed above (converting the constant “0” to an unconditional branch).

7.4 Maintaining Derivation Information

In our discussion of translation to tree form, there is one detail we have omitted which applies to all phases of translation: the maintenance of derivation information.

The Flex runtime system provides a copying garbage collector to perform memory-management for Flex-compiled programs. Unfortunately, the ability to perform unambiguous full copying collection imposes some stern low-level requirements upon the collector. From [11]:

> *i)* *it must be able to determine the size of heap allocated objects*
>
> *ii)* *it must be able to locate pointers contained in heap objects, so they can be both traced and updated.*
>
> *iii)* *it must be able to locate pointers in global variables.*
>
> *iv)* *it must be able to find all references in the stack and in the registers at any point in the program at which collection may occur.*
>
> *v)* *it must be able to find objects that are referred to by values created as a result of pointer arithmetic.*
>
> *vi)* *it must be able to update these values when the objects involved are moved.*

Of all of these requirements, perhaps the 5th is the trickiest. Often times, the compiler aliases a pointer to point to the interior of some object as the result of some optimization. Such a pointer is known as a *derived pointer.* These derived pointers are as important in a garbage collector's liveness analysis as non-derived (tidy) pointers. While the solution of all requirements is important, only the solution to the 5th has a substantial bearing on the compilation of tree form.

This solution employed by the Flex compiler is described in[11]. While a full discussion of this solution is beyond the scope of this paper, it will be necessary to cover the basics of the algorithm in order to understand its role in the translation process.

The algorithm proposed in [11] constructs a number of tables at compile time to assist in the garbage collection process. In particular, one set of tables is constructed at each point in the program where garbage collection can occur (called a gc-point). Each gc-point has 3 tables: *stack pointers*, *register pointers*, and *derivations*. The first two of these are simple: *stack pointers* contains a list of live non-derived on the stack frame, *register pointers* contains a list of non-derived pointers in registers. The third table describes the derivations of all derived pointer values at the gc-point.

The derivation of a pointer consists of a map of base locations to relations. For example, a pointer *a* might be derived like so:

```
a := b1 + b3 - b2 + E
```

where b1, b2, and b3 are derived values, and E is some non-derived integer expression. Then, the derivation table for a would be constructed like so:

Base Location	Relation
b1	+
b2	-
b3	+

Figure 9: A derivation table for "a".

To be able to generate the proper tables for garbage collection, it is necessary to be able to calculate derivation information for any derived pointers at every gc-point. To this end, the Flex compiler maintains maps of variables to their "derivation" lists, throughout all phases of compilation at the level of LowQuadSSA or lower. Thus, any translation pass which operates on LowQuadSSA or lower must take care to preserve these derivation lists.

While this maintenance of derivation information is not challenging, it is a detail that must be remembered during translation. There are many ways to preserve this information, but the remainder of this section will discuss the strategy employed in the translation to Tree form.

In the tree translation passes, each translator is supplied with a hashtable. The translator uses this hashtable to store the new derivation information that is created during the pass. This information is stored in key/value pairs where the key is a tuple consisting of an HCodeElement and a Temp, and the value is an instance of the harpoon.IR.Properties.Derivation.DList class, which is designed to store derivation information:

```
Derivation:  { HCodeElement, Temp } ---> DList
```

Derivation information must be added to the derivation tables when one of the following two events occurs:

1) *Derivation information about a temporary changes.*
2) *A new derived pointer is added to the representation.*

Once the translation pass completes, the updated derivation information can be accessed through the following algorithm:

```
Derivation(HCodeElement hce, Temp t) {
     Let newD be the table of new derivation
          information.
     Let oldD be the old derivation information, if it
          exists, null otherwise.
     If newD contains the key { hce, t }:
          return the value associated with that key.
     Else:
          If oldD is null, return null.
          Else: return oldD.Derivation(hce, t).
}
```

The nested nature of derivation information makes accessing this information less efficient than we might desire. However, the big advantage of this algorithm is that it potentially saves a great deal of memory by storing only the derivation “delta” across translation passes.

The derivation information preserved through this approach is exposed through the harpoon.IR.Properties.Derivation interface. In Tree form, all codeviews implement this interface, allowing access of derivation information directly from the codeview.

7.5 Maintaining Type Information

It is useful to know the type of non-derived temporaries in an IR. This is especially true in IRs which do not explicitly assign types to their elements, as Tree form does. It is nonetheless useful in Tree form, as the type information exposed directly by Tree form is too imprecise for some analyses, such as the creation of gc-tables.

Type information is maintained in a manner analogous to derivation information. In QuadSSA form, type information is computed by a type analysis algorithm. All subsequent passes add new type information to a hashtable, using the following mapping function:

```
TypeMap:  { HCodeElement, Temp } --->   HClass
```

Once the translation pass completes, the new type information can be accessed through an algorithm very similar to the access of derivation information. The one caveat associated with typemaps is that *it is an error to access the type of a derived pointer!* If you attempt such an access, an error will be thrown. Thus, accessing a type must be a two step process: first, you should check that no derivation information exists for the `Temp` in question. Only if the derivation information associated with the Temp is null, should you proceed to access the *type* of the `Temp`.

```
TypeMap(HCodeElement hce, Temp t) {
     If derivation information exists for { hce, t }:
          throw an error.
     Let newT be the table of new type information.
     Let oldT be the old type information if it exists,
          null otherwise.
     If newT contains the tuple { hce, t }:
          return the value associated with that key.
     Else:
          If oldT is null: return null.
          Else: return oldT.typeMap(hce, t).
```

```
    }
```

The type information preserved through this approach is exposed through the `harpoon.Analysis.Maps.TypeMap` interface. In tree form, all codeviews implement this interface, allowing access of type information directly from the codeview.

7.6 Canonical Tree Form

For the most part, the Tree form previously described is a good representation of a generic machine language. There is, however, one problem which makes it unfit for general use: the ESEQ expression.

The main problem with the ESEQ expression is that it makes different orders of tree evaluations yield different results. This makes analysis and translation much harder.

It may not be clear why this element of Tree form exists at all, given the trouble it causes. The reason it is permitted is that it simplifies the translation *to* Tree form from previous IRs [1]. Specifically, it allows us to nest expressions which require pre-computation, such as pointers to newly allocated memory, and the results of INSTANCEOF checks.

Fortunately, Appel has already implemented a translation pass to translate to canonical tree form, and the Flex compiler uses a modified version of it. (In this case, *modified* means *reduced*. The original Flex tree form is closer to canonical form than Appel's, so only parts of the transformation are necessary). The algorithm for converting to canonical tree form can be found in chapter 8 of [1], but we'll give an overview of the algorithm here.

The goal of the Flex canonicalization pass is simple: remove ESEQs from a tree while preserving the semantics of the tree. This is accomplished by moving all ESEQs to the top level subexpressions of a statement, and then inserting the "stm" subexpression of the ESEQ *before* the ESEQ's parent.

A recursive application of Appel's ESEQ identities is sufficient for this task, and can be found in [1].

8 Analysis and Optimization

At this time, Tree form performs two important optimizations: constant propagation and algebraic simplification. That algebraic simplification is performed at this level is unsurprising -- it doesn't require control or dataflow information, and tree form's nested expressions provide many opportunities for simplification. Constant propagation (or any other dataflow optimization) is not an optimization that generally lends itself to tree-based representations[2]. Sadly, it is necessary to perform constant propagation at this level, as the translator to Tree form generates some extra copy statements (this greatly simplified the translator).

The remainder of this section discuss the mechanisms for performing these analyses, and the methodology by which you can integrate them with your transformations.

8.1 Acquiring a Control-flow Graph

Constant propagation requires the use of a control flow graph. IRs in the Flex Compiler infrastructure with which a control-flow information can be associated must implement one of two interfaces: `harpoon.IR.Properties.CFGraphable`, or `harpoon.IR.Properties.CFGrapher`.

The first interface is implemented by representations which are inherently connected in a control-flow structure, such as the quad forms. In this case, the *elements* of the IR would implement `CFGraphable`.

The second interface is of greater interest to us, and is implemented by representation which are not inherently connected in a control-flow structure. The methods of this interface take in an element of the representation as a parameter, and return lists of predecessor or successor edges as a result. In Tree form, the codeview provides a method by which to acquire a `CFGrapher` object, which can then be used to externally associate control flow with the tree.

It is not immediate obvious how control flow can be associated with a tree. This association relies upon the following rules:

1) *The elements of the control flow graph for any given tree form consist of all of the statements in that tree form, EXCEPT for any node of type SEQ.*

2) *Two elements, x and y, of the control flow graph are connected if*

   a) *x directly precedes y by fall-through execution*

   b) *x branches directly to y*

Without rule 1, extracting control-flow information from a tree seems to make little sense. The inclusion of expressions in a control-flow graph would be confusing, hard to define consistently, and inefficient. The obvious inefficiencies would be increased space requirements for control-flow graphs, and increased time requirements for computing dataflow equations (although the time required to actually solve the dataflow equations would not be affected). Rule 2 simply provides the standard definition of a control flow graph.

Unfortunately, it is not always easy to tell if one node precedes another by fall-through execution in Tree form. This difficulty results from the specification of the SEQ node, which is used to define control flow in Tree form. Recall that the SEQ node specifies that its entire left side is executed before its right side. The result of this definition is that the predecessor of a statement, *s*, could be deeply nested in another tree far from *s*, making it impossible to determine the predecessor of a statement in isolation of the rest of the tree. Another difficulty arises in that the CJUMP and JUMP instructions in Tree form don't actually have pointers to the Trees which they branch to. They essentially store only the name of the label they branch to.

Both of these difficulties are solved by the `harpoon.IR.Tree.TreeGrapher` class, the tree forms implementation of `CFGrapher`. The `TreeGrapher` constructs a control-flow graph from a supplied tree using the following algorithm:

```
BuildCFG(Tree root) {
     Construct a map, M, between label names and the
          tree elements which they label.
     Let s be a new Stack.
     Let curr be root.
     while curr != null:
          if curr is a CALL statement:
               Let Dest1 be the location of exception
                    handling code for curr.
               Let Dest2 be s.pop().
               Add an edge between curr and Dest1.
               Add an edge between curr and Dest2.
               Let curr be Dest2.
          else if curr is a CJUMP statement:
               Let Dest1 be the destination of curr's
                    true branch.
               Let Dest2 be the destination of curr's
                    false branch.
               Using M, add an edge between curr and
                    Dest1.
               Using M, add an edge between curr and
                    Dest2.
               Let curr be s.pop().
          else if curr is a JUMP statement:
               Using M, add an edge between curr, and
                    each possible destination of curr's
                    computed branch.
               Let curr be s.pop().
          else if curr is RETURN or a THROW statement:
               Let curr be s.pop().
          else if curr is a SEQ statement:
               s.push(curr.right).
               Let curr be curr.left.
          else
               Let Dest1 be s.pop().
               Add an edge between curr and Dest1.
               Let curr be Dest1.
}
```

This external grapher can be acquired programmatically from any Tree codeview through a call to harpoon.IR.Tree.Code.getGrapher(). This call executes the BuildCFG algorithm on the elements of the Tree codeview upon which it is invoked. *Note:* the grapher does not dynamically update itself when the code from which it is acquired is modified! If you modify the tree code, you must obtain a new grapher if you require a correct control-flow graph. Each call to `getGrapher()` requires the recomputation of the control flow graph, so try not to call it too frequently.

8.2 Flex Dataflow Analysis Infrastructure

Correct constant propagation in a non-SSA representation requires a reaching definitions analysis [2].

The Flex DataFlow analysis infrastructure provides a convenient, easy-to-use interface to a set of dataflow analysis routines. However, a little study is necessary before one can effectively make use of this infrastructure.

DataFlow analyses in the Flex infrastructure are performed in three phases: 1) *The Construction of a Flowgraph of Nodes*, 2) *The Calculation of Flow Functions*, and 3) *Solving the Constructed Dataflow Equations.* While the Flex DataFlow Infrastructure is flexible enough to provide many different implementations of each of these phases, we will discuss the implementation of reaching definitions analysis, which uses the most common implementation techniques, and is employed by the Tree form. Further discussion of the dataflow analysis infrastructure is beyond the scope of this paper.

1) Construction of a Flowgraph of Nodes

This step consists of the construction of a set of maximal basic blocks from the elements of a codeview. The code to perform this operation resides in the `harpoon.Analysis.BasicBlock` class. The `BasicBlock` class can construct a graph of basic blocks from any `CFGrapher` (which is exposed by Tree codeviews).

2) Calculation of Flow Functions

This step is performed by the `harpoon.Analysis.DataFlow.ReachingHCodeElements` class. The function performed by this class is twofold: firstly, it initializes a mapping between the basic blocks computed in step #1, and current IN, OUT, GEN, and PRSV sets of the block[1], and secondly, it exports the *merge* and *transfer* functions for the analysis.

[1] For more discussion on these sets, turn to [2].

The *merge* function propagates dataflow information between adjacent basic blocks and the *transfer* function propagates the dataflow information associated with the IN set of a basic block to the OUT set. For reaching definitions analysis, these functions are:

Merge Function: $$IN(i) = \bigcup_{j \in Pred(i)} OUT(j) \forall i$$

Transfer Function: $$OUT(i) = GEN(i) \cup (IN(i) \cap PRSV(i)) \forall i$$

3) Solving the Flow Functions

Finally, the Flex dataflow equation solver (`harpoon.Analysis.DataFlow.Solver`) iteratively solves the dataflow equations formulated in step 2. After the dataflow equations have been solved, the results are exported through the `ReachingHCodeElements` class.

The benefit of this dataflow analysis infrastructure is the high amount of code reuse. In effect, it factors out much of the code required for all dataflow analyses, and allows only the specification of the analysis' dataflow functions to vary.

8.3 Constant Propagation

At last, we have enough information to describe the Tree constant propagation algorithm. Indeed, now that we have discussed the acquisition of a control flow graph, and the Flex dataflow analysis infrastructure, there is very little left to the algorithm [12]:

```
ConstantPropagation(Tree root) {
     Construct a CFG from root using a CFGrapher.
     Run a Reaching Definitions analysis on root.
     Foreach element, t, in the tree defined by root:
          if t is a TEMP expression:
               Let tDefs be the set of statements
                    which define t.
               Let rDefs be the set of definitions
                    which reach t.
               Let tReach be (tDefs intersect rDefs).
               If each element of tReach assigns some
                    constant value. K, to t:
                    Replace t with K.
}
```

When actually implemented, this algorithm would have to be modified slightly for efficiency reasons. For example, it would be more efficient to compute a mapping between temps and their corresponding definitions once at the start of the algorithm, rather than computing this value dynamically as the pseudocode would suggest.

8.4 Algebraic Simplification

This optimization is much more “appropriate” for a tree-based representation than is constant propagation, as it requires no dataflow or control-flow analysis, and can be most effective on trees because of the large expressions they contain.

The algebraic simplification pass works by iteratively applying a simplification function to each expression in the tree, until that expression reduces to a fixed point of the function. Each expression is guaranteed to reach a fixed point, because the simplification function is required to be monotonic.

Assuming that you do not specify your own set of rules, a default set of simplification rules is provided. These rules perform the following simplifications[2]:

1) *Simple Arithmetic Identities:*

Combining Constants:

```
k1 + k2   ---> k3
k1 * k2   ---> k3
k1 << k2  ---> k3
k1 >> k2  ---> k3
k1 >>> k2 ---> k3
k1 & k2   ---> k3
k1 | k2   ---> k3
k1 ^ k2   ---> k3
```

Commutative Property:

```
k + x      ---> x + k
k * x      ---> x * k
k & x      ---> x & k
k | x      ---> x | k
k ^ x      ---> x ^ k
```

Removing 0's:

```
x + 0      ---> x
x << 0     ---> x
x >> 0     ---> x
x >>> 0    ---> x
```

Removing negation:

```
-(-e))     ---> e
-0         ---> 0
```

2) *Conversion of multiplications to shifts.*

Any multiplication by a constant can be replaced by a series of additions and shifts [8]. The reason for this useful fact may not be immediately obvious, but it is actually quite simple. First, consider that any n-bit number can be programmatically represented like so:

```
x     =     1*(x&1)  + 2*(x&2)+ 4*(x&4)  + 8*(x&8)  ...
                 + 2^(n-1)*(x&2^(n-1))
```

We can therefore represent an arbitrary multiplication by a 32-bit integer constant, k, like so:

```
x * k = x *  (1*(k&1)  + 2*(k&2)  + ...  (2^31)*(k&(2^31)))

x * k = x*1*(k&1)  + x*2*(k&2)  + ... x*(2^31)*(k&(2^31))

x * k = x<<((k&1)*0)  + x<<((k&2)*1)  + ..
          x<<((k&(2^31))*31)
```

Finally, since k is constant, we can combine the multiplication in each term, leading to an expression of the form:

```
x * k = x<<K1  + x<<K2  + x<<K3  + ...  + x<<KN
```

While this naive reduction certainly replaces the multiplication with shifts, we can do better. The current algorithm reduces a multiplication to N shifts, and N-1 additions, where N is the number of bits set to "1" in k. While we cannot improve this algorithm, we can reduce the number of bits set to "1" in our constant using an old trick called a Booth Recoding[8].

Booth recoding works by converting "strings" of 1's in a binary number to a subtraction of two numbers with fewer ones. For example:

```
01111111 = 10000000  - 00000001
```

Lets say we were converting a multiplication by 01111111 to a series of shifts and additions. Performing a booth recoding first would save 5 shifts and 6 additions at the expense of 1 subtraction! The algorithm for performing a booth recoding can be found in many texts, including [8].

Thus, our final algorithm consists of 1) performing a Booth recoding upon the constant expression, and 2) converting a multiplication by the resulting expression to shifts and adds, as described above.

3) *Conversion of divisions to multiplications*

Just as multiplications by constants can be converted into similar operations for increased efficiency, so can divisions by constants. In *Division by Invariant Integers*[13], Granlund and Montgomery outline an algorithm for doing just that.

Unlike the algorithm for converting multiplications to shifts, the algorithm for converting divisions to multiplications is complex, and is beyond the scope of this paper. However, a full discussion of the suggested implementation can be found in [13].

The algorithm converts 1 division to 4 bitwise operations, 3 additions, and 1 multiplication.

8.4.1 Using the Algebraic Simplification Optimizer

The interface to the algebraic simplification routines in tree form is very simple. It consists of two methods in the `harpoon.Analysis.Tree.AlgebraicSimplification` class:

```
public static void simplify(Stm root);
public static void simplify(Stm root, List rules);
```

Both methods simplify `root` in place. The first method uses the default set of rules described above. The second method allows you to specify a set of simplification rules. Each rule you define must implement the `AlgebraicSimplification.Rule` interface:

```
public static interface Rule {
     public boolean match(Exp e);
     public Exp apply(Exp e);
}
```

The `match()` method must return true if the supplied expression matches the rule. The `apply()` method returns the application of the rule to its parameter. For example, the following would be a rule to remove extraneous additions by 0:

```
Rule removeZero = new Rule()  {
     public boolean match(Exp e) {
          if (e.kind() == TreeKind.BINOP) {
               BINOP b = (BINOP)e;
               Exp right = b.getRight();
               if (right.kind() == TreeKind.CONST)
                    CONST c = (CONST)right;
                    return c.getValue().intValue()==0;
               }
          }
          return false;
     }

     public Exp apply(Exp e) {
          BINOP b = (BINOP)e;
          return b.getLeft();
     }
};
```

This rule could then be used to simplify the tree form using the following code:

```
List rules = new ArrayList();
rules.add(removeZero);
AlgebraicSimplification.simplify(myTree, rules);
```

While the ability to define new rule sets improves the flexibility of this simplification pass, one should use it sparingly. First, it is easy to mistakenly create a list of functions which does not monotonically reduce expressions. Secondly, the new rules would likely be less inefficient than the default rules. The default rules use a special set of bitmasks which allow for the creation of efficient `match()` methods.

9 Future Improvements

While the tree form has proved to be an effective IR for the Tree Form, there is still some work left to be done on it.

9.1 Removal of Non-canonical Trees as Codeviews

Like much of the Flex compiler in this early stage, the tree translation routines run slowly. This is probably due to the large number of object creations involved in constructing the Tree form. Thus, one possible improvement might be to remove non-canonical tree as a codeview. This would have the following benefits:

1) *The elements of non-canonical tree code would not have to be cloned*

The current conversion to canonical tree form is forced to clone each entire non-canonical view before translation. Since tree codes are generally very large, this is a time-consuming operation. Removing non-canonical tree code as a codeview means that it can be directly modified in the translation to canonical Tree form, allowing for a potentially substantial speedup.

2) *The elements of non-canonical tree code would not need to be maintained*

The current implementation of the Flex compiler keeps each representation of code it produces in memory. Removing non-canonical tree code as a codeview means one less codeview that will be stored in memory.

3) *Non-canonical tree code should not be used anyway*

It is difficult to correctly analyze and modify non-canonical tree code anyway. Hiding its existence might be a good way to prevent tricky bugs from creeping into analysis passes.

9.2 On-the-fly Tree Folding

Currently, redundant subexpressions created by the tree translation pass have to be optimized out by a separate tree folding pass. However, [2] suggests that a more sophisticated translation pass can be used to prevent the creation of these redundant subexpressions in the first place.

The main benefit arising from this improvement is increased computational efficieny. As previously discussed, Tree form is not an appropriate IR for dataflow analyses; they tend to be rather slow and inefficient. Therefore, the replacement of a dataflow analysis pass with a more complex initial translation could lead to a substantial speedup.

9.3 Testing

If you've written a nice optimization pass for the Tree form, how do you know if it is correct? One possible method is through the use of a Tree interpreter: an interpreter that interprets Tree form directly.

Of course, having the Tree interpreter successfully interpreter one's code is not a guarantee of correctness, but it is a good start. An interpreter could also be used for regression testing, to help catch new bugs quickly before they get lost in the shuffle.

Correctness is more important than efficiency, a fact it is sometimes easy to forget.

10 Final Thoughts

Despite its inefficiencies, using a tree-based low-level IR has proved to be a good decision for the Flex Compiler Infrastructure. Tree form's retargetability has made it relatively easy to support multiple platforms, and its orthogonality and simplicity have made it easy to construct and optimize.

This comes as no surprise -- trees are widely used as low-level IRs, and with good reason.

11 References

[1] A. Appel. 1997. *Modern Compiler Implementation in Java.* Cambridge University Press, Cambridge, United Kingdom.

[2] S. Muchnick. 1997. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, San Francisco, CA.

[3] The Flex Compiler Documentation. Available: http://www.flex-compiler.lcs.mit.edu/Harpoon/doc/index.html

[4] E. Gamma et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading MA.

[5] Sethi, Ravi, J. Ullman. The Generation of Optimal Code for Arithmetic Expressions, *JACM.* Vol. 17, No. 4, Oct.1970, pp. 715-728.

[6] W. Stevens. 1992. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading MA.

[7] D. Elsner, J. Fenlason, et al. 1994. *Using as: The GNU Assembler.* Available: http://www.gnu.org/manual/gas-2.9.1/html_mono/as.html

[8] R. Paul. 1994. *SPARC Architecture, Assembly Language Programming, & C.* Prentice Hall.

[9] W3C Document Object Model (DOM) Level 2 Specification, Version 1.0. Available: http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210/

[10] J. Gosling, B. Joy, G. Steele, et al. 1996. *The Java Language Specification.* Addison-Wesley, Reading MA.

[11] A. Diwan, E. Moss, R. Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.* pp. 273-282.

[12] M. Rinard, S. Amarasinghe. Dataflow Analysis and Traditional Optimizations. *MIT 6.035 Course Notes.* Available: http://web.mit.edu/course/6/6.035/6035.html

[13] T. Granlund, P. Montgomery. Division by Invariant Integers using Multiplication. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* pp. 61-72.