## TABLE OF CONTENTS

## 1 Introduction

The Smalltalk-80* programming language has a number of advantages. It allows systems to be built quickly. It provides good support for building interactive programs and good support for graphics. Classes and class inheritance provide modularity and promote code reuse. It has a simple model of computation, since everything in the system is an object and all computation consists of messages passed from one object to another. The user can access every part of the system. Smalltalk-80 provides a highly graphical interface and most objects have a visual representation. It provides a rich programming environment [Goldberg 84] that promotes code reuse and rapid prototyping.

Smalltalk's major shortcoming is that it is less efficient than traditional programming languages. This has often been blamed on the fact that each operation on an object is performed by sending a message to it. When a message is passed to an object, the Smalltalk interpreter searches the class of the object for a method, i.e. a procedure, corresponding to the message and executes that method. If no method is found, an error is generated. Since many classes may implement methods corresponding to a single message, the method to be executed in response to a message cannot be determined until the class of the receiver is known. Since variables are not typed in Smalltalk, the class of the receiver is not known until run-time, when the variable containing the receiver has a value.

*Smalltalk-80 is a trademark of Xerox Corporation

Attempts have been made to increase the performance of Smalltalk. These have concentrated on increasing the speed of method lookup through the use of sophisticated caching techniques [Conroy 83] [Deutsch 83], building more efficient interpreters [Caudill 86], or building special purpose hardware [Lewis 86]. Several attempts to build a compiler for Smalltalk-80 have been made [Atkinson 86][Ballard 86][Samples 86]. Still, Smalltalk's performance does not come close to that of more conventional programming languages. Further increases in performance will probably require building an optimizing compiler [Johnson 86].

An optimizing compiler for Smalltalk requires type information, and one particular version is Typed Smalltalk [Johnson 86][Johnson 87b][Graver 87]. This thesis, part of an ongoing project to build a compiler for Typed Smalltalk, describes how to perform high level optimizations to the parse tree generated by the compiler. It eliminates method lookup and procedure calls, generally considered to be costly operations, and converts the parse tree into a directed graph from which more efficient code can be generated. This graph is then converted into machine code [Wiegand 87]. Other parts of the compiler are described in [Johnson 87a] and [Whitledge 87].

This thesis consists of three major parts. The first part is a brief description of Typed Smalltalk. The second is a description of the parse trees produced by the compiler. The third part describes the optimizations made to the parse tree. These include elimination of runtime method lookup, elimination of non-recursive procedure calls through in-line substitution, elimination of tail-recursive procedure calls, and constant folding involving blocks.

## 2 The Language

The language we are using is a version of Smalltalk-80 with type information. The type system was developed by Ralph Johnson and Justin Graver and is described in detail in [Johnson 86] and [Johnson 87b]. Smalltalk-80 is described in [Goldberg 83].

Johnson's type system differs from earlier type systems like those introduced for Quicktalk [Ballard 86] and Hurricane [Atkinson 86] in that it maintains much of the power of Smalltalk. These earlier type systems did not allow polymorphism and so were unable to compile many Smalltalk methods. Other type systems were unable to provide enough information for optimization [Borning 82][Suzuki 81]. Our type system avoids both of these problems.

A type must be declared for every variable in each class and method. In addition, a return type for each method must be specified. The return type can usually be inferred by the type-checker.

The type of a single object is an instance of **ObjectType**. An **ObjectType** contains the class of the object and possibly the types of components of the object. Examples of **ObjectTypes** are <**Integer**>, which contains only the class of the object, and <**Array of: String**>, which contains not only the class of the object but also the type of each element of the array. **ObjectTypes** that include a list of types describing the components of the object are called *parameterized types*. In the case of a parameterized type, such as <**Array of: String**>, it is not sufficient to just specify that the object is an array. That would not give the type-checker or the optimizing compiler enough information about the object.

Smalltalk is polymorphic, i.e. a single variable might contain objects in many types. Thus, several **ObjectTypes** can be combined into a **UnionType**. When a variable's type is an instance of **UnionType**, e. g. <**String** + **Integer**>, it means that at runtime the variable can take on a value that is an instance of either **String** or **Integer**. The variable can, at different times during runtime, take on the value of several different types, each an **ObjectType** in its **UnionType**.

There are other classes of types, such as **SignatureType**. These are described in [Johnson 86] and [Graver 87]. They will not be considered here since they do not allow any optimizations by the compiler.

## 3 The Parse Tree

### 3.1 Creating Parse Trees

A Typed Smalltalk method is compiled by creating an instance of **TypedSmalltalkCompiler** and sending it the message compile:in:to:notifying:ifFail:. This message returns an unoptimized parse tree.

Typed methods are normally compiled by classes, which will create an instance of **NewCompiledMethod** that contains the parse tree and other information about the method. Thus, the easiest way to inspect a parse tree is to examine a **NewCompiledMethod**.

**NewCompiledMethod** is a subclass of **CompiledMethod**. Because of this, **NewCompiledMethod** inherits all of the information normally held in a **CompiledMethod**, including bytecodes. This lets the interpreter behave normally on typed methods.

```
CompiledMethod typedSubclass: #NewCompiledMethod
    instanceVariables: 'substitute <BooleanType>
                        dependents <OrderedCollection of: CompiledMethod>
                        isDependentOf <OrderedCollection of: CompiledMethod>
                        parsetree <TypedMethodNode>
                        optimized <BooleanType>
                        exec <ByteArray>
                        accessProtect <Semaphore>'
    classVariables: ”
    typeParameters: ”
    poolDictionaries: ”
    category: 'Typed-Support'
```

The instance variables in a **NewCompiledMethod** contain information needed to perform optimization of the method. These are explained below.

| | |
|---|---|
| **substitute** | A boolean value indicating whether in-line substitution of the method is allowed. |
| **dependents** | The set of compiled methods dependent on the method. Whenever the method is changed, every method in this set must be recompiled. |
| **isDependentOf** | The set of compiled methods on which the method is dependent. Whenever a method in this set is changed, the method must be recompiled. |
| **parsetree** | The typed parse tree for the method. |
| **optimized** | A boolean value indicating whether the method has been optimized. |
| **exec** | A byte array containing the executable code for the method. |
| **accessProtect** | A semaphore to ensure mutual exclusion during optimization. |

### 3.2 Components of Parse Trees

The parse tree created by the parser is composed of instances of subclasses (or subclasses of subclasses) of **TypedParseNode**. The structure of the tree is maintained by instance variables of each node which point to other nodes or collections of nodes.

Context-free grammars specify the syntax of languages. Parse trees specify the syntactic structure of sentences in a language. Parse trees have long been used to represent the syntactic structure of a program during compilation. They are considered of little value beyond this and are often omitted from compilers that generate intermediate code as syntax analysis is performed. Those compilers that do generate parse trees often use them as a form of intermediate language representation that can be optimized [Aho 86].

Attribute grammars, described in [Reps 84], can represent more than just syntactic information about a language. An attribute grammar is a context-free grammar that associates a set of attributes to each symbol in the grammar. Attributes can specify information about items in the language or relationships between items in the language. In this way, semantic information can be specified using an attribute grammar.

The information specified by an attribute grammar can be stored in the parse tree. The parse tree used in the Typed Smalltalk compiler is such a tree. In addition to specifying the syntactic structure of the method, the tree specifies information about each item in the method, such as type.

Each node in the parse tree has a method which specifies how to typecheck, generate code, print a representation of the node, perform data flow analysis, and optimize. These methods act like rules in an attribute grammar, specifying how attributes are propagated throughout the tree.

In a parse tree generated from an attribute grammar, there are two kinds of attributes. *Inherited attributes* are attributes that are passed down the tree. *Synthesized attributes* are passed up the tree. The arguments to the methods of parse nodes act like inherited attributes while the values returned by these methods act like synthesized attributes.

There are some differences between the Typed Smalltalk parse trees and the parse trees of attribute grammars. Typed Smalltalk is not limited by the functional properties of attribute grammars. That is, nodes can have methods unrelated to attribute passing. Typed Smalltalk doesn't have the automatic propagation of attributes associated with attribute grammars. Inherited attributes must be explicitly passed to other nodes as arguments to methods. Synthesized attributes must be explicitly returned as the value of a method.

The parse tree of an unoptimized compiled method can be made up of instances of the following subclasses of **TypedParseNode**.

| | |
|---|---|
| **TypedAssignmentNode** | An assignment statement. The instance variable variable contains a leaf node representing the left side of the assignment, while value contains a subtree representing the right side of the assignment. |
| **TypedBlockNode** | A block. The instance variable arguments contains a set of leaf nodes representing the formal parameters to the block. The instance variable statements contains a set of subtrees representing the statements in the body of the block. |
| **TypedCascadeNode** | A sequence of messages sent to a single receiver. The subtree representing the receiver, which can be an expression of any complexity, is stored in the instance variable receiver. A set of **TypedMessageNodes** representing the sequence of messages is in the instance variable messages. |
| **TypedCoercionNode** | A change to the type of an expression. This node is transparent to the optimization process. |
| **TypedMessageNode** | A message send. The subtree representing the receiver is in receiver, the string representing the selector of the message is in selector, and the set of expressions passed as actual parameters is in arguments. |
| **TypedMethodNode** | The root of the tree. The set of leaf nodes representing the formal parameters to the method is in arguments and the subtree representing the body of the method is in statement. |
| **TypedPrimitiveNode** | A system primitive. The instance variable failCase contains the subtree performed if the primitive fails. |
| **TypedProcedureNode** | A procedure call. This represents a message send for which the class of the receiver is known. The instance variables receiver, arguments, and selector contain the information about the message send. |
| **TypedReturnNode** | A return statement. The instance variable expr contains the subtree representing the expression to be returned. It is possible (and not uncommon) to have more than one **TypedReturnNode** in a single parse tree. |

| | |
|---|---|
| **TypedStatementListNode** | A list of sequential statements. The instance variable statements contains a collection of subtrees representing the statements. |

There is only one instance of **TypedMethodNode** in each parse tree. There can be any number of instances of the other classes.

Generally, **TypedProcedureNodes** are created by the optimization process (see Chapter 4). **TypedProcedureNodes** can appear in the parse tree before optimization, though. When the compiler encounters a message send to the receiver super, it converts it into a **TypedProcedureNode**, since the class in which the method is found is known.

The leaves of the parse tree can consist of instances of any of the following subclasses of **TypedLeafNode**.

| | |
|---|---|
| **TypedArgVarNode** | An argument to a method. |
| **TypedBlockArgNode** | An argument to a block. |
| **TypedContextNode** | The context on top of the stack. |
| **TypedGlobalVarNode** | A global variable. |
| **TypedInstVarNode** | An instance variable of a class. |
| **TypedLiteralNode** | A literal string or a number. |
| **TypedSelfNode** | The pseudo-variable self. |
| **TypedSpecialVarNode** | The pseudo-variables nil, true, or false. |
| **TypedTempRegVarNode** | A temporary variable created by the compiler. |
| **TypedTempVarNode** | A local variable of a method. |

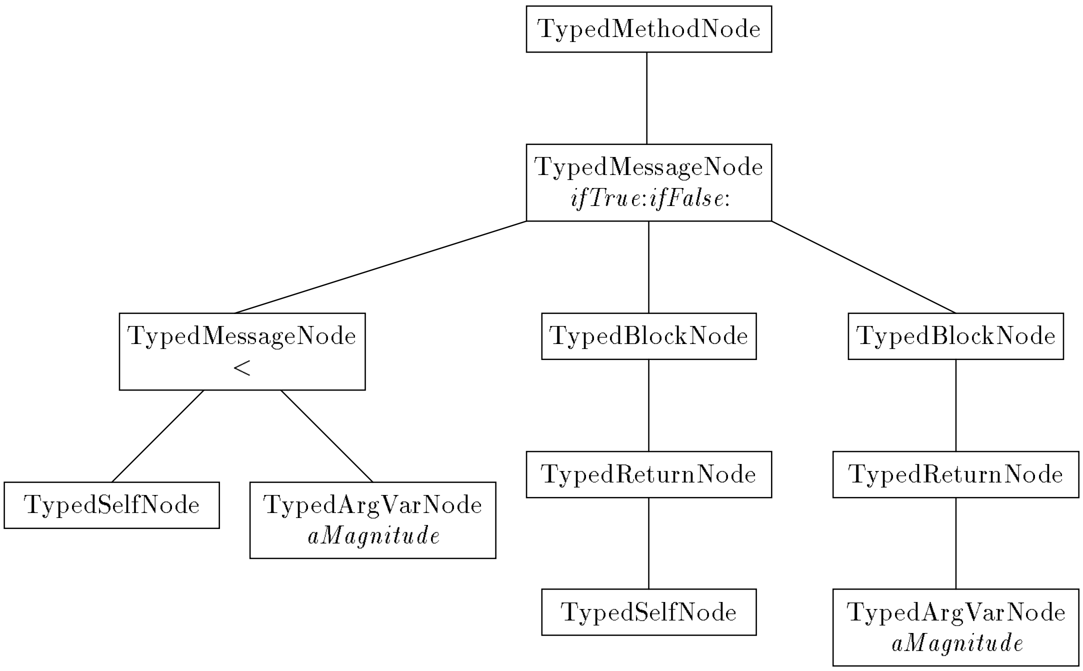


Figure 3.1: Parse tree for `min:` method in Magnitude

As an example, consider the parse tree for the method `min:` for the class **Magnitude**, shown in Figure 3.1.

```
{ arguments: aMagnitude <Self>
  returnType: <Self> }

min: aMagnitude
"Answer the receiver or the argument,
whichever has the lesser magnitude."

self < aMagnitude
    ifTrue: [↑self]
    ifFalse: [↑aMagnitude]
```

Although the parse tree generated by the typed parser will be constructed of instances of the above classes, the optimized graph will look very different. The optimizer often creates additional nodes during the course of optimizing the tree. Often these nodes simplify the structure of the tree.

The real purpose, however, is to allow more efficient code to be generated from the final directed graph. An optimized graph will be constructed of instances of the above classes plus instances of the following subclasses of **TypedParseNode**.

| | |
|---|---|
| **TypedCaseNode** | A case statement. |
| **TypedChoiceNode** | One kind of possible choice in a case statement. |
| **TypedChoiceClassNode** | Another kind of possible choice in a case statement. This is actually a subclass of **TypedChoiceNode**. |
| **TypedIndexNode** | An instance variable of an object other than the receiver. |
| **TypedJumpDestinationNode** | The destination of a jump. |
| **TypedJumpNode** | A jump to a **TypedJumpDestinationNode**. |

### 3.3 Operations on Parse Trees

Each parse node knows how to print itself, typecheck itself [Johnson 86][Johnson 87a][Graver 87], and generate code for itself [Wiegand 87]. This thesis describes how each node can optimize itself and perform in-line substitution on itself.

In-line substitution involves replacing a function call with the body of the function and replacing the formal parameters in the body with the actual parameters. Smalltalk passes arguments by value and the pass-by-value property must be maintained after the substitution. Thus the arguments to the function must be evaluated once before evaluating any of the statements in the body. The values of the arguments will replace the corresponding formal parameters in the body of the function.

Some parameters are safe to substitute directly. Items such as variables and literals, which require no computation, can be folded into the body without changing the functionality. Blocks fit into this category. Evaluating a block causes the creation of a block context. Blocks have no local variables, so two blocks created in the same context with the same body will have the same results when evaluated. The statements in the body of a block are not executed until the block is sent the value message, in which case it is no longer represented by a **TypedBlockNode** but by a **TypedMessageNode** with a **TypedBlockNode** as its receiver. Substitutions of these nodes directly sometimes allow further optimizations to be performed.

Substitution of a block could cause incorrect code to be generated in some cases. Consider the example where a block is passed as an argument to a function and then compared to itself as in the following method.

```
{ returnType: <Self>}

strange: aBlock

aBlock == aBlock
    ifFalse: [self error: "That is strange."]
```

The comparison of aBlock with itself should be true. If in-line substitution is performed on this method, however, the code for the block will be substituted for each aBlock. At runtime two different blocks with identical code will be compared, resulting in the error: message being sent. Such code is contrived and will occur rarely, so the problem is ignored for now. This needs to be considered in future versions, though.

An expression cannot be substituted directly if, like message sends and statement lists, its value might change when it is evaluated more than once. Such an expression must be evaluated once before the statements in the body of the substituted function and the value used during substitution.

To achieve this, two messages, makeLeafNodePartIn: and asLeafNode, are defined for each kind of parse node. A node that represents an expression with possible side effects responds to the makeLeafNodePartIn: message by returning a new **TypedAssignmentNode** with a new **TypedTempRegVarNode** as its variable and the original node as its value. This represents a statement that evaluates the expression and stores its value in a temporary variable. This statement can be placed in the tree before the body of the substituted function, ensuring that the expression will be evaluated before any of the statements in the body and providing a variable, the **TypedTempRegVarNode**, which can replace the formal parameter everywhere it appears.

A node that has no side effects, i.e. a node that can be substituted directly, responds to makeLeafNodePartIn: by returning itself. In addition, a **TypedAssignmentNode** returns itself. Although repeated evaluations of a **TypedAssignmentNode** could cause different results, there is no reason to create an assignment statement and a variable to hold the value, since these already exist. Therefore, the **TypedAssignmentNode** is returned with no changes and its variable is used for substitution.

Once makeLeafNodePartIn: has been sent to a node, sending the message asLeafNode to the result returns the variable or expression that can be substituted. **TypedAssignmentNodes** return the node on the left side of the assignment statement. Other nodes can be substituted directly and just return themselves. Notice that asLeafNode does not need to know whether makeLeafNodePartIn: changed the original node.

The message evaluateArgumentOn:in: is sent to the receiver and arguments of the procedure call before the procedure call is substituted in-line. Figure 3.2 shows how each argument (and the receiver) is converted to a leaf node using the message makeLeafNodePartIn:. If an assignment statement is created, it is placed at the beginning of the statements in the body of the procedure. The leaf node corresponding to this argument (or receiver) is returned to be substituted for the formal parameter of the procedure during in-line substitution.

```
evaluateArgumentOn: statements in: encoder
"The receiver is an argument to a procedure which has statements
as its body. Prepare the receiver for substitution into the
statements. That is, if the receiver is an expression, create an
assignment statement which will evaluate it and assign its value
to a temporary variable. Add the assignment statement to the
beginning of the statements and return the temporary variable.
If the receiver is not an expression, i.e. it has no side effects,
leave it alone."

| aNode |

aNode ← self makeLeafNodePartIn: encoder.
(aNode ~~ self) | (aNode asLeafNode ~~ aNode) ifTrue: [statements add: aNode].
↑ aNode asLeafNode
```

Figure 3.2: Evaluate arguments during in-line substitution.

# 4 Performing Method Lookup at Compile Time

## 4.1 Motivation

Smalltalk's dynamic method lookup is often blamed for much of its inefficiency. Paradoxically it is also one of its greatest strengths. The Smalltalk environment consists of objects which interact by passing messages. Each time a message is passed to an object, the message name is mapped to a procedure, i.e. a method, defined in the class of the receiver of the message, which is then executed. This allows a single message to correspond to many different methods, each in a different class. This contributes to Smalltalk's modularity and polymorphism.

In an untyped Smalltalk system, looking up the method must be performed at runtime. Since the receiver of the message can be practically any value, the system cannot look up the method until it knows the value of the receiver.

However, if the class of the receiver is known to be one of a small number of classes, the compiler can look up the methods corresponding to these classes at compile time. At runtime, when the class of the receiver is known, no method lookup need be performed, only a case statement to choose the proper method.

Dynamic method lookup is an obvious inefficiency and is a costly operation in many interpreters. Caching of method lookups has proven successful in removing much of the inefficiency associated with method lookups [Conroy 83][Deutsch 83]. This adds some complexity to the interpreter, of course, but largely eliminates the cost of method lookup.

```
shouldCaseConvertIn: encoder
| aType |

aType ← receiver typeIn: encoder.
↑aType isObject or: [aType isUnion and: [aType size < 3]]
```

Figure 4.1: Determine whether compile-time method lookup should be done for a TypedMessage-Node.

Compile time method lookup does more than eliminate the runtime method lookup, though. It allows us to do many optimizations on the code that are impossible in the dynamic system. These are described in the later chapters.

### 4.2 Determining When to Case Convert

The addition of a type system to Smalltalk makes it possible to determine the set of possible classes to which an object can belong at compile time. By the time a method is optimized, type-checking has already occurred. The type of each node is available, so method lookup can be performed at compile time.

Whenever a **TypedMessageNode** is encountered during optimization, it is sent the message shouldCaseConvertIn:, shown in Figure 4.1, to determine whether it should be converted into a case statement of procedure calls. Right now, a message node is converted if its receiver has a type that consists of only one or two classes, i.e. at runtime the receiver could only be an instance of one of two or fewer possible classes. This number is arbitrary and changing it might result in better optimized code.

It might also result in better optimized code if other things were considered to determine whether to look up a message at compile time. For instance, if a particular message send is performed often, perhaps if it is inside a loop, it may be better to convert it into a case statement even if the receiver of the message could be an instance of one of several possible classes. On the other hand, a message send that is seldom performed is less important to convert, even if the receiver is of one or two possible types. More complicated flow analysis will need to be done in order to implement this.

### 4.3 Performing Case Conversion

Once it has been determined that a **TypedMessageNode** should be converted to a case statement, the message caseConversionIn:inReturn: is sent to the **TypedMessageNode**. The first argument to this message is a list of encoders. The first encoder always points to the method being optimized. The others point to methods that have been substituted. The second argument tells whether this message send is the value of a return statement. It is used in tail-recursion elimination (chapter 6).

The caseConversionIn:inReturn: method in **TypedMessageNode**, shown in Figure 4.2, converts a message send into a case statement of procedure calls. First, it converts the receiver of the message into a leaf node so it does not have to be evaluated for each choice in the case statement. Then, a **TypedProcedureNode** is created for each class in the type of the receiver. The **TypedProcedureNode**, which contains the following instance variables, contains all the information needed to perform the procedure call.

| | |
|---|---|
| **receiver** | The receiver of the message. This is the result of sending the message makeLeafNodePart: to the original receiver. It is either the original receiver of the message or an assignment of the receiver to a temporary variable to avoid evaluating the receiver more than once. |

```
caseConversionIn: encoders inReturn: inReturn
| procNode cases aClass tmpReceiver tmpCase |

receiver ← receiver makeLeafNodePartIn: encoders first.
tmpReceiver ← receiver asLeafNode.
cases ← (receiver typeIn: encoders first) typeSet asOrderedCollection collect: [:atype |
            aClass ← atype baseClass.
            procNode ← TypedProcedureNode
                                   receiver: tmpReceiver
                                   arguments: arguments
                                   selector: selector
                                   type: "The return type of this procedure."
                                   class: (aClass whichClassIncludesSelector: selector).
            aClass isUnique
               ifTrue: [TypedChoiceNode
                            choice: (encoders first encodeVariable: aClass soleInstance printString)
                            trueCase: procNode]
               ifFalse: [TypedChoiceClassNode
                            choice: (encoders first encodeVariable: aClass printString)
                            trueCase: procNode]].
tmpCase ← (TypedCaseNode testCase: tmpReceiver options: cases)
                  optimizeIn: encoders inReturn: inReturn.
↑receiver == tmpReceiver
                    ifTrue: [tmpCase]
                    ifFalse: [TypedStatementListNode with: receiver with: tmpCase]
```

Figure 4.2: Perform compile-time method lookup for a TypedMessageNode.

| | |
|---|---|
| **arguments** | The arguments passed along with the message. |
| **selector** | The name of the message. |
| **class** | The class in which the method can be found. |
| **retType** | The type returned by the method. |

The instance variable class contains the class in which the method corresponding to the message can be found. This is not necessarily the class of the receiver. Each class inherits the methods of its superclasses and the method can be defined in any class along the hierarchy. The message whichClassIncludesSelector:, when sent to a class and passed a selector, looks up the hierarchy starting at the class and returns the first class in which a method with the selector is defined.

Notice that the **TypedProcedureNode** does not contain the actual method, just information needed to find the method. It is easy to find the method once the class in which it exists is known. However, if the actual method were stored, the method we are compiling would be dependent on it. A change to any method stored in a procedure node in the method would require the method to be recompiled.

If only one procedure node is created, i.e. the receiver can be in only one possible class, the **TypedProcedureNode** replaces the original **TypedMessageNode** in the tree. Otherwise a **TypedCaseNode** must be created. It consists of the following instance variables.

| | |
|---|---|
| **testCase** | The value of the receiver of the message send. This has already been converted to a leaf node so the receiver will not be evaluated more than once. |
| **options** | The set of choices from which a procedure is chosen at runtime. |
| **otherwise** | A statement which is executed if the value of the receiver does not match any of the options. |

Each element of options is an instance of either **TypedChoiceNode** or **TypedChoiceClassNode**. There is one option for each **TypedProcedureNode** created. If the class in which the method is found is a class that has only one instance, for example, **True**, **False**, or **UndefinedObject**, an instance of **TypedChoiceNode** is created. It has the following instance variables.

| | |
|---|---|
| **choice** | The sole instance of the class in which the method is found. |
| **trueCase** | The **TypedProcedureNode** that is performed if, at runtime, the value of testCase equals choice. |

**TypedChoiceClassNode** is the more general case. It is a subclass of **TypedChoiceNode** and has no instance variables of its own. The instance variable choice that it inherits from **TypedChoiceNode** contains the class in which the method is found. At runtime, the class of testCase is compared with the value in choice. If they are the same class, the procedure call in trueCase is performed.

The **TypedCaseNode** is returned to replace the original **TypedMessageNode** in the tree. At runtime the receiver of the message is evaluated. The case statement chooses the proper procedure call to be performed. The method is accessed, passed the value of the receiver and the arguments of the message send, and evaluated.

## 5 In-line Substitution of Procedure Calls

### 5.1 Motivation

An important optimization in procedural languages is in-line substitution, the replacement of a procedure call by the code of the procedure [Scheifler 77]. This can also be important in Smalltalk, since a message send is essentially a method lookup followed by a procedure call. It was just shown that method lookup can be done at compile time using case conversion. Once the case conversion has determined the method (or group of possible methods) corresponding to a message send, we have access to the parse tree for the method. Since we also know the receiver and arguments of the message send, we can fold the values of each into the method, replacing the procedure call with the parse tree for the method. This has two advantages. First, at runtime a procedure call is avoided and, although not a particularly costly operation, it isn't free. Second, the folding in of the arguments and receiver may allow us to perform further optimizations, such as constant folding, on the substituted nodes.

We don't want to do this indiscriminately, however. Substituting every procedure call would cause several problems.

- The size of the parse tree (and therefore our final machine code) could increase exponentially.
- Each method is dependent on all methods that are substituted into it. Every time a method is changed, each method in the system into which that method has been substituted will have to be recompiled.

**shouldSubstituteInto: encoders**

```
self substitute == nil
   ifTrue: [self substitute: false.
            ↑false]
   ifFalse: [substitute
               ifTrue:
                 [encoders do:
                    [: enc | (enc contextClass compiledMethodAt: enc method selector)
                    == self ifTrue: [↑false]].
                  ↑true]
               ifFalse: [↑false]]
```

Figure 5.1: Determine whether in-line substitution should be done for a NewCompiledMethod.

- If there is a procedure call to the method we are compiling or a method that was substituted earlier, in-line substitution of this procedure would result in an infinite recursion of substitutions.

## 5.2 Determining When to In-Line Substitute

A **NewCompiledMethod** responds to the message shouldSubstituteInto: with a boolean indicating whether it should be replaced by the method it calls. The definition of shouldSubstituteInto: is shown in Figure 5.1. In-line substitution is allowed if the procedure call is

1. not a recursive call and
2. has been marked as substitutable by the user.

The shouldSubstituteInto: message needs a list of methods to determine if a procedure call is recursive. A procedure call is recursive if it is a call to the method being optimized or to any of the previously substituted methods within which this procedure call exists. That is, after a procedure call has been substituted, its parse tree is optimized. While optimizing this subtree, a procedure call to the method that was substituted or to the method being compiled is a recursive call. If it were substituted, since substitution is followed by optimization of the substituted parse tree, the same procedure call would be encountered and substituted, and this would continue infinitely (or until the parse tree grew too large for memory). Extending this, it is clear that the optimization process can be deep inside several nested substitutions. Therefore, in order to determine whether a procedure call is recursive, the call must be compared to each substituted method along the path from the node representing the call to the root of the tree. A list of **TypedEncoders** is kept around to allow access to these methods.

An instance of **TypedEncoder** exists for each method in the system. It functions as a local symbol table for the method, storing information about the method's literal frame, local variables, and type information. The argument to shouldSubstituteInto: is a list of encoders for every method substituted along the path to this **TypedProcedureNode**. If the method of the **TypedProcedureNode** is the same as the method of any of these encoders, it is a recursive call and therefore unsubstitutable.

There are many other reasons a method should not be substituted. The current parse tree may be too large already. The parse tree of the method being considered for substitution may be too large. The method may be changed often, causing every method into which it was substituted to be recompiled.

It is clear that determining these conditions is difficult. To facilitate testing and development of the compiler, the user determines whether a non-recursive procedure call is suitable for substitution by specifying the value for the variable substitute in each method. This is clearly not sufficient for later versions of the compiler and it is an area in which more work will have to be done.

Even more complicated analysis could result in better optimized trees. [Ball 82] suggests several considerations before a procedure is substituted. Among these is the frequency with which the procedure call is performed. For instance, if the flow of control almost never reaches a procedure call then the method is hardly worth substituting even if it is small and unchangeable. A larger method, however, may be worth substituting if its procedure call is often reached, if it is in an inner loop, for example.

### 5.3 Performing In-line Substitution

In-line substitution is started in the optimizeIn:inReturn: method for **TypedProcedureNode**, shown in Figure 5.2. Once it has been determined that in-line substitution should be done on a method, dependency information is updated. If the method being substituted is an inherited method, i.e. it is not defined in the class of the receiver, a copy of it is installed in the class of the receiver to aid in maintaining dependency pointers. The dependency mechanism is described further in [Whitledge 87]. The message substitute:for:receiver:class:inMethod:jumpDestination: (hereafter referred to as the substitute message) is sent to the parse tree for the method of the procedure call. The message propagates throughout the tree, i.e. each node sends the message to its subtree, returning a copy of the parse tree with the arguments to the procedure call substituted in place of the actual parameters of the method and the receiver of the procedure call substituted in place of the variable self. The parse tree is then optimized with the encoder of its root added to the list of encoders so that recursive calls to this method can be recognized.

Each node of the method being substituted must create a new node to return as the value of the substitute method. The new node replaces the old node in the parse tree. Since in-line substitution is started at the root of the parse tree being substituted and propagates throughout the tree, the entire parse tree is copied and replaces the **TypedProcedureNode** representing the call to the method being substituted. If the tree of the substituted method were not copied, modifications during substitution would change the original parse tree of the substituted method, which is clearly not what is wanted.

The **TypedMethodNode** that is the root of the parse tree to be substituted is sent the substitute message to start the in-line substitution. The method that corresponds to this message is shown in Figure 5.3. The argument method is only used by the **TypedMethodNode** and is therefore not sent to the other nodes in the tree. The substitute:for:receiver:class:jumpDestination: message will be referred to as the substitute message also.

In-line substitution consists of three steps. First, the method being substituted adds local variables to the method into which it is substituted. Second, it evaluates each argument and the receiver, making a list of the resulting values. Finally, these values are substituted into the body of the method, which replaces the procedure call in the parse tree. Each of the steps is discussed in more detail in the following paragraphs.

The local variables, temporaries, of the method being substituted must be combined with the temporaries of the method being compiled. One might think that aliasing problems would arise if both methods have a temporary with the same name. This problem is avoided because we are representing variables as objects in the parse tree. Two temporary variables may have the same name but they will be different objects. Since the names are only used for printing, aliasing is not a problem.

```
optimizeIn: encoders inReturn: inReturn
| subMethod subParseTree subEncoder type newEncoders |

"Code to process constant folding."
      ⋮
subMethod ← class compiledMethodAt: selector.
subEncoder ← "Nil if the call isn't recursive."
subEncoder isNil
ifTrue: [(subMethod shouldSubstituteInto: encoders)
     ifTrue:
          [type ← receiver typeIn: encoders first.
          (type isObject and: [type baseClass ~= class])
               ifTrue: [subMethod ← CopyCompiledMethod
                                        copy: subMethod
                                        selector: selector
                                        class: type baseClass].
          subMethod addDependent: (encoders first contextClass compiledMethodAt:
                                       encoders first method selector).
          subParseTree ← subMethod parseTree.
          newEncoders ← encoders copy.
          newEncoders addLast: subParseTree encoder.
          ↑ (subParseTree
               substitute: arguments
               for: nil
               receiver: receiver
               class: class
               inMethod: encoders first method
               jumpDestination: (TypedJumpDestinationNode method: encoders first method
                                                   type: (self typeIn: encoders first)))
               optimizeIn: newEncoders inReturn: inReturn]
     ifFalse: [inReturn ifTrue: [↑TypedReturnNode expr: self]
                         ifFalse: [↑self]]]
ifFalse: [... "Code for tail-recursion elimination." ...]
```

Figure 5.2: Start in-line substitution for a TypedProcedureNode.

```
substitute: args for: args2 receiver: rcvr class: aClass inMethod: method
jumpDestination: aLabelNode
| stmts newargs mainStatement newArg newRcvr i |

"Combine the temporaries of the substituted method with mine."
method addTemporaries: temporaries in: encoder.

"Evaluate the receiver and the arguments."
stmts ← TypedStatementListNode new.
newargs ← args collect: [:arg | arg evaluateArgumentOn: stmts in: method encoder].
newRcvr ← rcvr evaluateArgumentOn: stmts in: method encoder.

"Start the in-line substitution."
mainStatement ← statement substitute: newargs
                          for: arguments
                          receiver: newRcvr asLeafNode
                          class: aClass
                          jumpDestination: aLabelNode.

"All returns have been converted to jumps. Add a label after the
statements and optimize the case where the only return is at the end."
stmts add: (mainStatement addTrailingJumpIfNeeded: aLabelNode).
↑stmts
```

Figure 5.3: Substitution of a TypedMethodNode.

The arguments and receiver of a message can be expressions of arbitrary complexity. Arguments in Smalltalk are passed by value. Thus the body of the method being substituted is preceded by a list of assignment statements that evaluate the arguments and receiver and assign the values to new **TypedTempRegVarNodes**. These **TypedTempRegVarNodes** are later substituted in place of the formal parameters of the procedure. If an argument is already in its simplest form (i.e. evaluation would not change it), it is not included in the assignment statements, but is substituted directly. Substituting simple arguments directly makes further optimizations easier. For example, substitution of a **TypedBlockNode** allows constant folding if the block is later sent the value message (Chapter 7).

Return statements in the substituted method should return to the point in the original method immediately following the call to the substituted method. This point is labeled with a **TypedJumpDestinationNode** at the end of the substitute method for **TypedMethodNode**, as shown in Figure 5.3. As the substitute message propagates throughout the parse tree of the substituted method, all return statements in the body of the substituted method are replaced with jumps to this point. Figure 5.4 shows how a return statement is converted to an assignment of the expression to a temporary variable and then a jump to the **TypedJumpDestinationNode** marking the end of the method.

After the substitute message has propagated throughout the tree of the substituted method, the substitute method for **TypedMethodNode** places the **TypedJumpDestinationNode** in the tree immediately following the body of the substituted method. This **TypedJumpDestinationNode** acts as a label marking the spot to which the original procedure call returned. All return statements in the body of the substituted method have been replaced with jumps to this point.

```
substitute: values for: variables receiver: rcvr class: aClass jumpDestination: aLabelNode
"Return a jump to the statement following the original procedure call."

↑TypedStatementListNode with: (TypedAssignmentNode
                                    variable: aLabelNode result
                                    value: (expr substitute: values
                                                 for: variables
                                                 receiver: rcvr
                                                 class: aClass
                                                 jumpDestination: aLabelNode))
                        with: (TypedJumpNode to: aLabelNode).
```

Figure 5.4: Substitution of a TypedReturnNode.

```
substitute: values for: variables receiver: rcvr class: aClass jumpDestination: aLabelNode
"Replace me with the value passed to the method."

variables with: values do: [:var :val | key == var key ifTrue: [↑val]].
self error: 'Could not find argument to substitute myself'
```

Figure 5.5: Substitution of a TypedArgVarNode.

Replacement of a formal parameter by its actual parameter takes place at the leaf nodes of the tree. A **TypedArgVarNode** responds to the substitute message by returning the leaf node of the corresponding actual parameter, thereby replacing the formal parameter with the value of the argument passed it. This method is shown in Figure 5.5.

## 6 Tail-Recursion Elimination

### 6.1 Motivation

Smalltalk has no built-in iteration statements or *goto* statements. Therefore, iterative operations such as do: or whileTrue: are defined using recursion. While other message sends can be eliminated using in-line substitution, as described in Chapter 5, recursive procedures were not substituted in-line. Thus, iterative operations will be costly unless recursive procedure calls can be optimized.

Sometimes the very last thing a method does is a message send and the value returned by the method is the value returned by the message send. This is called a *procedure call in tail position.* It is costly and unnecessary to save the state of a method making a procedure call in tail position. The procedure call can be converted to a jump to the beginning of the called method.

A recursive procedure call in tail position is called *tail-recursive.* A tail-recursive procedure call can be optimized by replacing it with a jump to the section of the tree that represents the method being called. Iterative operations are usually defined using tail-recursion. Optimizing tail-recursive procedure calls will not only make iterative operations fast but will sometimes make other programs more efficient, too.

### 6.2 Detecting Tail-Recursive Procedure Calls

Tail-recursive calls can arise in three ways:

- The value of a return statement is a procedure call to the method being compiled.

```
whileTrue: aBlock

↑self value
   ifTrue: [aBlock value.
            self whileTrue: aBlock]
```

Figure 6.1: One implementation of whileTrue:.

- A procedure call in tail position that was substituted (see chapter 5) has a return statement whose value is a procedure call either to the method being compiled or the method that was substituted. Since return statements in substituted methods that are not in tail position are converted to jumps, any such procedure call is in tail position with respect to the method being optimized and is therefore tail-recursive. Extending this, a procedure call that is nested within several substituted methods is tail-recursive if each substituted method was in tail position and the procedure call is a call to any of the substituted methods.

- A return statement's value is an arbitrarily complex expression that has a recursive procedure call in a tail-recursive position. That is, the last thing to be done in the expression is the procedure call. Notice that flow of control does not have to always reach the procedure call, but if it does, the value of the procedure call is the value returned by the return statement. An example of this is the method whileTrue: as written in figure 6.1.

The first argument to the optimizeIn:inReturn: message is an ordered collection of **TypedEncoders**. When optimization is started, this ordered collection contains only the **TypedEncoder** of the method being compiled. When a method has been substituted in-line, the **TypedEncoder** corresponding to its **TypedMethodNode** is added to the ordered collection while its subtree is

```
optimizeIn: encoders inReturn: inReturn
| subMethod subParseTree subEncoder type newEncoders |

"Code to process constant folding."
      ⋮
subMethod ← class compiledMethodAt: selector.
subEncoder ← inReturn ifTrue: [subMethod isTailRecursiveIn: encoders].
subEncoder isNil
ifTrue: [... "Code for In-line substitution." ...]
ifFalse: [↑self installJump: subEncoder method initialLabel in: subEncoder]
```

Figure 6.2: Eliminate tail-recursion in a TypedProcedureNode.

being optimized. The functional properties of our optimization ensure that when we return from optimization of this subtree, we will continue the optimization of the rest of the tree with the list of **TypedEncoders** in effect before the substitution was made. Thus the list of **TypedEncoders** will grow as the optimization process travels down the tree performing substitutions and optimizing the subtrees that are substituted, and shrink as the process returns back out of these subtrees.

In order to detect tail-recursion of the first two types, each time the optimizeIn:inReturn: message is sent to a **TypedProcedureNode** that is inside a **TypedReturnNode**, the message isTailRecursive is sent to the compiled method that the procedure node represents (see figure 6.2). This will determine whether the call is to any of the methods that have been substituted earlier along the path between this node and the root node or to the method being compiled. The method isTailRecursive is shown in figure 6.3.

It is a difficult problem to detect tail-recursion of the third type. A better solution is to reduce all tail-recursive calls to calls of the first two types. To accomplish this, we push all return statements as far into the method as they will go, so that no return statements have complex expressions being

**isTailRecursiveIn: encoders**

```
encoders do: [:enc | (enc contextClass compiledMethodAt: enc method selector)
                    == self ifTrue: [↑enc]].
↑nil
```

Figure 6.3: Detect a tail-recursive NewCompiledMethod.

**optimizeIn: encoders inReturn: inReturn**

```
↑expr optimizeIn: encoders inReturn: true.
```

Figure 6.4: Optimize a TypedReturnNode.

returned. For example, the return statement

```
↑self value
    ifTrue: [aBlock value.
            self whileTrue: aBlock]
    ifFalse: [nil]
```

can be rewritten as

```
self value
   ifTrue: [aBlock value.
           ↑self whileTrue: aBlock]
   ifFalse: [↑nil].
```

The return statements are pushed as far as they can go by a boolean value that is passed as the second argument of optimizeIn:inReturn:. This boolean value, initially false, is set to true while the expression inside a **TypedReturnNode** is being optimized. As shown in Figure 6.4, the **TypedReturnNode** is replaced by the node returned by optimizeIn:inReturn:.

```
optimizeIn: encoders inReturn: inReturn
| newStatements |

newStatements ← statements collect: [:each | each == statements last
   ifTrue: [each optimizeIn: encoders inReturn: inReturn]
   ifFalse: [each optimizeIn: encoders inReturn: false]].
↑TypedStatementListNode statements: newStatements
```

Figure 6.5: Optimize a TypedStatementListNode.

```
optimizeIn: encoders inReturn: inReturn
"Nodes for which there is no optimization possible will
default to this message which will just return the node
with no optimization done"

inReturn
  ifTrue: [↑TypedReturnNode expr: self]
  ifFalse: [↑self]
```

Figure 6.6: The default optimization method for all parse nodes.

Complex nodes, such as **TypedStatementListNodes**, pass the flag along as the optimizeIn:inReturn: message is propagated to the node's components, as shown in Figure 6.5. Simple nodes that cannot be optimized further, such as leaf nodes, respond to optimizeIn:inReturn: by returning a **TypedReturnNode** with the receiver as the value of expr, as shown in Figure 6.6. Note that a **TypedReturnNode** with a simple expression is replaced in the tree with a **TypedReturnNode** identical to itself.

Once all returns have been pushed as deep into the tree as they can go, only tail-recursive calls of the first two types exist and are easily detected.

### 6.3 Eliminating Tail-Recursion

If a procedure call is tail-recursive, isTailRecursiveIn: returns the encoder of the method being called. The **TypedMethodNode** of this method contains a **TypedJumpDestinationNode**, i.e. a label at the beginning of the method. The tail-recursive **TypedProcedureNode** is replaced in the tree by a **TypedJumpNode** to this **TypedJumpDestinationNode**, effectively replacing the tail-recursive procedure call with a jump to the beginning of the method. The code for this is in Figure 6.2.

# 7 Constant Folding

## 7.1 Motivation

If a compiler knows the value of a variable at compile time, it can replace the variable with its value. Likewise, when the values of the arguments to an expression are known at compile time, the values can be folded into the expression, i.e. each occurrence of an argument in the expression can be replaced by its value.

Many compilers perform constant folding on arithmetic expressions, array accesses, and other simple expressions. Typed Smalltalk does too, but these low level optimizations are performed in a later phase of optimization [Wiegand 87]. However, constant folding on blocks is done during high level optimization on parse trees.

In Smalltalk, a block is used to delay execution of a section of code. The block is a normal object and, as such, can be passed as an argument to procedures, i.e. blocks or methods, and can be returned as the value of a procedure. The body of the block will be executed only when the block is passed the `value` message, or, if the block has one, two, or three arguments, the `value:`, `value:value:`, or `value:value:value:` message. All four of these will be referred to as the `value` message.

When the compiler encounters a block being sent the `value` message, the block and the message send can be eliminated and replaced with the statements in the body of the block with the arguments to the block substituted into the block. Since a block is used to delay execution of code, it is rare that (and hard to understand why) a programmer would write code on which we could perform this optimization. In-line substitution, however, will cause this kind of code to occur frequently.

As an example, consider compiling and optimizing the method abs for **Number**. The type <**Self**> refers to the type of the receiver of the message.

```
{ returnType: <Self>}

abs
"Answer a Number that is the absolute value
(positive magnitude) of the receiver."

self < 0
    ifTrue: [↑self negated]
    ifFalse: [↑self]
```

Since the receiver (self < 0) of the message can only be in one of two classes, **True** or **False**, the message send would be converted to a case statement (chapter 4). A temporary variable, **t1**, is created by the optimizer to avoid multiple evaluations of the receiver.

```
{ temporaries: t1 <BooleanType>
  returnType: <Self>}

abs
| t1 |

t1 ← self < 0.
Case t1
    true: call True:(ifTrue: [↑self negated] ifFalse: [↑self])
    false: call False:(ifTrue: [↑self negated] ifFalse: [↑self])
```

Notice that, since there is only one instance each of **True** and **False**, the test statement of the case statement is compared with the sole instance of each class.

The method ifTrue:ifFalse: for the class **True** is defined as

```
{ localTypeVariables: <T> <ReturnType>
  arguments: trueBlock <Block of: ReturnType>
             falseBlock <T>
  returnType: <ReturnType>}

ifTrue: trueBlock ifFalse: falseBlock
    ↑trueBlock value
```

and for the class **False** as

```
{ localTypeVariables: <T> <ReturnType>
  arguments: trueBlock <T>
             falseBlock <Block of: ReturnType>
  returnType: <ReturnType>}

ifTrue: trueBlock ifFalse: falseBlock
    ↑falseBlock value
```

Neither of these methods are recursive and both are substitutable. Therefore in-line substitution (chapter 5) will be performed and the following code will result.

```
{ temporaries: t1 <Boolean>
  returnType: <Self>}

abs
| t1 |

t1 ← self < 0.
Case t1
    true: [↑self negated] value
    false: [↑self] value
```

At this point, constant folding will be done in order to avoid a block creation and a message send. The result is

```
{ temporaries: t1 <Boolean>
  returnType: <Self>}

abs
| t1 |

t1 ← self < 0.
Case t1
    true: ↑self negated
    false: ↑self
```

Two message sends and the creation of two blocks were optimized away. In practice, the receiver of the message, self < 0, would be optimized as well. This was omitted to simplify the example.

```
optimizeIn: encoders inReturn: inReturn
| subMethod subParseTree subEncoder type newEncoders |

arguments ← arguments collect: [:arg | arg optimizeIn: encoders inReturn: false].
(receiver class = TypedBlockNode and: [selector == #value
                                        or: [selector == #value:
                                        or: [selector == #value:value:
                                        or: [selector == #value:value:value: ]]]])
    ifTrue: [↑self constantFolding: inReturn].
    ⋮
"Code to process tail-recursion elimination and in-line substitution."
    ⋮
```

Figure 7.1: Start constant folding for a TypedProcedureNode.

## 7.2 Performing Constant Folding

A constant block that is evaluated should be replaced by the statements in the body of the block. This happens during optimization, when a **TypedProcedureNode** recognizes that its receiver is an instance of **TypedBlockNode** and its selector is one of the four value messages. The constantFolding: message is sent to the **TypedProcedureNode** as shown in Figure 7.1.

The method constantFolding:, shown in Figure 7.2, will replace the **TypedProcedureNode** with the statements in the body of the block. In order to fold the arguments into the block, the statements are preceded by a list of assignment statements assigning the arguments to the parameters of the block.

```
constantFolding: inReturn
"Fold arguments in to a block, returning the statements in the body
of the block. The body is preceded by a list of assignment statements
which assign the arguments to their corresponding formal parameters.

| node child list |

node ← TypedStatementListNode new.
receiver arguments with: arguments do:
    [:var :val |
    node add: (TypedAssignmentNode variable: var value: val)].
node addAll: receiver statements.
inReturn ifTrue:
    [child ← node statements removeLast.
    node statements addLast: (TypedReturnNode expr: child)].
↑node
```

Figure 7.2: Perform constant folding on a TypedProcedureNode.

# 8 Conclusions and Future Work

## 8.1 Future Work

This thesis covers part of a project to build an optimizing compiler for Typed Smalltalk in an attempt to increase the performance of Smalltalk. The primary goal of this thesis is to add high-level optimizations to the compiler in order to eliminate the most costly inefficiencies of Smalltalk code. There is still much work to be done in this area and on the compiler itself.

In order to facilitate the optimizations performed, some simplifying assumptions were made. Specifically, more flow analysis should be performed in order to determine when compile-time method lookup and in-line substitution should be done. At this time, compile-time method lookup is done if the receiver of the message is an instance of fewer than three possible classes. This is sufficient for our testing of the compiler but there is no reason to believe that it is giving us optimal code. It would be more reasonable to base the decision of whether to lookup a message at compile time on the following:

- The number of possible classes of which the receiver can be an instance.
- The likelihood of a receiver being an instance of one of its possible classes. For example, a variable may be of type <**SmallInteger + UndefinedObject**>, but it may only be an instance of **UndefinedObject** rarely.
- The frequency with which the message send is performed.

Likewise, deciding when to in-line substitute now is done simplistically. A procedure call is substituted in-line if it is not a recursive call and if the user specifies it as being substitutable. This is sufficient for testing but in a final production of the compiler, the compiler should decide whether to substitute a procedure call based on information such as the following:

- Whether the procedure call is a recursive call.
- The current size of the parse tree.
- The size of the parse tree being considered for substitution.
- How often the method corresponding to the procedure call is changed.
- How often the flow of control reaches the procedure call.

### 8.2 Conclusion

This thesis has demonstrated that with the addition of a type system to Smalltalk, many of the operations that have generally been considered inefficient can be optimized away by the compiler. The code that is generated from an optimized parse tree can be expected to be much more efficient than that generated from an unoptimized parse tree.

The thesis has also demonstrated that the optimizations that need to be done to an object-oriented programming language like Smalltalk are often quite different than those that appear in compilers for more conventional languages.

## Bibliography


[Aho 86] Aho, Alfred C., Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Atkinson 86] Atkinson, Robert G. "Hurricane: An Optimizing Compiler for Smalltalk." Special Issue of *SIGPLAN Notices* (OOPSLA '86 Proceedings), Volume 21, Number 11, November 1986, pp. 151–158.

[Ball 82] Ball, John E. "Program Improvement by the Selective Integration of Procedure Calls." Ph.D. thesis, University of Rochester, Rochester, New York, 1982.

[Ballard 86] Ballard, Mark B., David Maier, and Allen Wirfs-Brock. "QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods." Special Issue of *SIGPLAN Notices* (OOPSLA '86 Proceedings), Volume 21, Number 11, November 1986, pp. 140–150.

[Borning 82] Borning, Alan H. and Daniel H. H. Ingalls. "A Type Declaration and Inference System for Smalltalk." *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982, pp. 133–139.

[Caudill 86] Caudill, Patrick J. and Allen Wirfs-Brock. "A Third Generation Smalltalk-80 Implementation." Special Issue of *SIGPLAN Notices* (OOPSLA '86 Proceedings), Volume 21, Number 11, November 1986, pp. 119–130.

[Conroy 83] Conroy, Thomas J. and Eduardo Pelegri-Llopart. "An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations." *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983, pp. 239–247.

[Deutsch 83] Deutsch, L. P. and A. M. Schiffman. "Efficient Implementation of the Smalltalk-80 System." *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 1983, pp. 297–302.

[Goldberg 83] Goldberg, Adele and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Goldberg 84] Goldberg, Adele. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[Graver 87] Graver, Justin O. "Adding Type Specification and Type Checking Capabilities to Smalltalk-80." M.S. thesis, University of Illinois, Urbana, Illinois, 1987.

[Johnson 86] Johnson, R. E. "Type-Checking Smalltalk." Special Issue of *SIGPLAN Notices* (OOPSLA '86 Proceedings), Volume 21, Number 11, November 1986, pp. 315–321.

[Johnson 87a] Johnson, Ralph E. "A Type System for Smalltalk: An Optimizer's View." Department of Computer Science, University of Illinois, Urbana, Illinois, 1987.

[Johnson 87b] Johnson, Ralph E. "A User's Guide to Typed Smalltalk." Department of Computer Science, University of Illinois, Urbana, Illinois, 1987.

[Lewis 86] Lewis, David M., David R. Galloway, Robert J. Francis, and Brian W. Thomson. "Swamp: A Fast Processor for Smalltalk-80." Special Issue of *SIGPLAN Notices* (OOPSLA '86 Proceedings), Volume 21, Number 11, November 1986, pp. 131–139.

[Reps 84] Reps, Thomas William. "Generating Language-Based Environments." Ph.D. thesis, Cornell University, Ithaca, New York, 1983.

[Samples 86] Samples, A. Dain, David Ungar, and Paul Hilfinger. "SOAR: Smalltalk Without Bytecodes." Special Issue of *SIGPLAN Notices* (OOPSLA '86 Proceedings), Volume 21, Number 11, November 1986, pp. 107–118.

[Scheifler 77] Scheifler, Robert W. "An Analysis of Inline Substitution for a Structured Programming Language." *Communications of the ACM*, Volume 20, Number 9, September 1977, pp. 647–654.

[Suzuki 81] Suzuki, Norihisa. "Inferring Types in Smalltalk." *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, 1981, pp. 187–199.

[Whitledge 87] Whitledge, James R. "An Interface for an Optimizer in the Highly Interactive Environment of Smalltalk." M.S. thesis, University of Illinois, Urbana, Illinois, 1987.

[Wiegand 87] Wiegand, John D. "An Object-Oriented Code Optimizer and Generator." M.S. thesis, University of Illinois, Urbana, Illinois, 1987.